High-Level Cryptographic Abstractions

Christopher Kane  
Stony Brook University  
ckane@cs.stonybrook.edu

Bo Lin  
Stony Brook University  
bolin@cs.stonybrook.edu

Saksham Chand  
Stony Brook University  
schand@cs.stonybrook.edu

Scott D. Stoller  
Stony Brook University  
stoller@cs.stonybrook.edu

Yanhong A. Liu  
Stony Brook University  
liu@cs.stonybrook.edu

ABSTRACT

The interfaces exposed by commonly used cryptographic libraries are clumsy, complicated, and assume an understanding of cryptographic algorithms. The challenge is to design high-level abstractions that require minimum knowledge and effort to use while also allowing maximum control when needed.

This paper proposes such high-level abstractions consisting of simple cryptographic primitives and full declarative configuration. These abstractions can be implemented on top of any cryptographic library in any language. We have implemented these abstractions in Python, and used them to write a wide variety of well-known security protocols, including Signal, Kerberos, and TLS.

We show that programs using our abstractions are much smaller and easier to write than using low-level libraries, where size of security protocols implemented is reduced by about a third on average. We show our implementation incurs a small overhead, less than 5 microseconds for shared key operations and less than 341 microseconds (< 1%) for public key operations. We also show our abstractions are safe against main types of cryptographic misuse reported in the literature.

1 INTRODUCTION

Existing cryptographic libraries are difficult to use. They require expertise that most developers lack, and place tedious burdens on experienced developers. The danger posed by this difficulty increases due to the proliferation of distributed computing, which requires that more developers make more extensive use of cryptographic libraries. The difficulty of using these libraries is not new [4, 70], but the extent of the problem grows as computers become more deeply embedded in our everyday lives. Mobile computing and the Internet of Things (IoT) especially encourage the storage and transmission of sensitive data. Consumers will expect such data to be properly secured, requiring that developers make greater use of cryptographic libraries.

A symptom that reveals the underlying problem with existing cryptographic libraries is the issue of cryptographic misuse—improper use of cryptographic APIs leading to violations of security requirements [35, pp. 73]. A string of papers, beginning in 2012, have documented widespread cryptographic misuse in mobile applications [9, 24, 35, 37, 45, 47, 52, 75]. These works define specific types of cryptographic misuse and build tools to detect them.

For example, Egele et al. [35] found that 88% of the 11,748 Android apps analyzed by their tool, CryptoLint, contained at least one cryptographic misuse. A similar study by Ma et al. [52] found that 99% of the 8,640 Android apps they analyzed contained at least one cryptographic misuse. However, detecting misuse is not sufficient to prevent all misuse.

At the same time, higher-level cryptographic libraries have been developed, as discussed in Section 7, but they limit the choices that experts can make in using and experimenting with different cryptographic algorithms. What are the right high-level abstractions that avoid different pitfalls?

This paper proposes SecAlgo, high-level abstractions for cryptographic operations that aim to minimize the difficulty of using cryptographic libraries in writing secure programs, for both non-experts and experts. SecAlgo provides simplest cryptographic primitives plus full declarative configurations. It can be implemented on top of any cryptographic library in any language. We have implemented SecAlgo in Python, and used it to write a variety of well-known security protocols, including Signal [68], Kerberos [61], and TLS [31].

We show that SecAlgo reduces programmer effort and increases clarity of programs; implementation of the abstractions incurs minimum overhead; and the abstractions prevent significant types of cryptographic misuse.
We first describe the need for better cryptographic abstractions in Section 2. We define SecAlgo abstractions in Section 3, and describe their implementation in Section 4. Section 5 shows application in implementing security protocols. Section 6 presents experimental results. Section 7 discusses related work and concludes. Appendix A presents studies of types of cryptographic misuse prevented.

2 WHAT ARE THE RIGHT CRYPTOGRAPHIC ABSTRACTIONS?

There are two reasons that better abstractions for cryptographic operations are needed: (1) APIs for existing low-level cryptographic libraries make it too difficult to properly use cryptographic primitives and (2) existing high-level libraries with simplified interfaces place too many restrictions on what experts can choose, because of the limited expressive power of the abstractions used.

Difficult-to-use low-level cryptographic libraries.

The low-level APIs provided by cryptographic libraries require many decisions, including ones that must be coordinated, sometimes repeatedly, in order to correctly use cryptographic primitives. Making these decisions demands significant expertise and tedious manual effort. Even experienced programmers are likely to make mistakes. Such mistakes will compromise the security objective the programmer tries to achieve by using the cryptographic primitives, as studied in the cryptographic misuse literature, e.g., [24, 35, 52, 75].

Proper use of low-level cryptographic APIs requires considerable security expertise. So one might think that we can address the difficulty of using these APIs by providing clear, thorough documentation and high-quality examples of proper use. However, it will not suffice as a solution, because the real issue is the sheer number of low-level decisions that must be made when using low-level APIs.

A right solution must directly address this complexity by protecting both non-experts and experts from mistakes caused by carelessness, neglect, limited time and attention, exhaustion, and other conditions that afflict humans confronted with unnecessary complexity.

Unduly restrictive high-level cryptographic libraries.

There are now a number of higher-level cryptographic libraries, Tink [5], Libsodium [38], Keyczar [78], and pycrypto's Fernet API [71], that provide simplified interfaces to reduce the tedious difficulty of using cryptographic functionality and assist non-expert users to avoid mistakes. This is accomplished by offering a limited set of abstractions for common operations and a restriction on the choice of algorithms, key sizes, and other configuration options.

However, expert users can find the limitations imposed by these high-level libraries restrictive and confining. Experts may want to use or test particular combinations of algorithms in security protocols and secure applications. They may find it difficult or impossible to do this when using existing high-level cryptographic libraries.

This issue may force expert users to return to low-level cryptographic libraries, because the large number of arguments, options, and configurations made available by low-level cryptographic libraries affords expert users their desired flexibility and expressive power.

SecAlgo: High-level abstractions with full range of control.

What’s needed is a way to abstract over cryptographic functionalities that combines the desirable features of both low-level and high-level cryptographic libraries while avoiding their drawbacks. We want to provide a simple, safe interface for all users to easily use the cryptographic functionalities they need, while giving expert users the expressive power to configure the cryptographic functionalities they want.

SecAlgo provides such high-level abstractions, requiring minimum, or zero, control by non-expert users, and allowing maximum, or full, control by expert users. In SecAlgo, the abstractions for determining what cryptographic operation to perform are pulled apart from the abstractions used to determine how that operation is performed.

- SecAlgo includes five primitive functions: keygen for key generation, encrypt and decrypt for public key and shared key encryption, and sign and verify for public key signing and MAC creation. MAC is treated as a kind of signing, just as in Bellare et al. [10]. Cryptographic hash functions are also included as a degenerate form of signing (if no key is supplied, sign returns a hash). These are all the basic cryptographic operations that any application might require. Every function except for keygen takes only the data and key as arguments.

- The user can optionally specify the encryption or signing algorithm to use when generating the key, otherwise a safe default algorithm is used. Non-expert users can use all the functionalities they need in a safe and simple way.

- SecAlgo also provides a rich and expressive abstraction for declarative configuration. Expert users can declaratively configure all attributes used by all cryptographic algorithms to exert fine-grained control over the behavior of SecAlgo’s primitive functions. In this way, SecAlgo provides the flexibility that expert users desire without reintroducing the complexity of low-level cryptography libraries.

Other important cryptographic functionalities, such as key management functions, can be added to SecAlgo by building them on top of the five primitive functions or by wrapping existing implementations of the functionalities in cryptographic libraries.

3 HIGH-LEVEL CRYPTOGRAPHIC ABSTRACTIONS

The abstractions in SecAlgo are of two kinds: (1) high-level cryptographic operations to provide confidentiality, integrity, and authenticity, and (2) declarative configuration for all feasible combinations of options. We describe language constructs for the first kind along with their essential arguments; additional arguments can be specified either as optional arguments or using configuration. Properly chosen configuration values—as determined by the current best practices in security—are used as default.

SecAlgo abstractions aim to serve both experts and non-experts. Experts can use optional arguments and configuration to call and
compose a wide range of cryptographic operations. Non-experts can rely on the default arguments and configurations to use safe cryptographic operations.

**Abstractions for cryptographic operations.**

SecAlgo has five basic cryptographic operations for writing security protocols and secure applications: key generation, encryption, decryption, signing, and verification. SecAlgo provides an abstraction for each basic operation. The abstractions allow simplest use of cryptographic algorithms—only the choice of shared vs. public key algorithms needs to be made for key generation, and only the data and key are needed for the other four operations.

All other choices are provided through declarative configurations. Users can configure multiple interrelated arguments among a set of choices for each argument, and the state configured at key generation is maintained through subsequent, continued uses of the other four operations. This contrasts with operations, such as generating a random value, that return only a simple value to be used subsequently.

We use a generic syntax in this section; each of the five operations can be implemented in any programming language as a single API function.

**Key generation** is of the following form, where type \( t \) is the name of a particular cryptographic algorithm, such as AES or RSA, or a generic shared or public.

\[
\text{keygen type } t
\]

It generates and returns a key or pair of keys suitable for the cryptographic algorithm corresponding to type \( t \).

- If \( t \) is the name of a specific shared-key (also called symmetric-key) algorithm, such as AES, or is the generic shared, then `keygen` returns a single value, a shared key. This key will be labeled with the name of the algorithm, \( t \). If the generic shared type is used, then the key is labeled with the name of the configured shared-key algorithm.
- If \( t \) is the name of a specific public-key (also called asymmetric-key) algorithm, such as RSA, or is the generic public, then `keygen` returns a pair of values, a private key and a public key. Each key is labeled with the name of the algorithm, \( t \). The two components of the pair can be retrieved by using a simultaneous assignment (in languages that support such assignments, such as Python) or by using two retrieval operations.

The size of the key can be specified as an optional argument to `keygen`, in an additional clause `size s` or otherwise declared as part of the configuration; if the size is left unspecified the default is used.

If \( t \) names a block cipher, such as AES, then a mode of operation [32]—an algorithm for repeatedly applying the block cipher to encrypt an arbitrary size plaintext, such as Cipher Block Chaining (CBC)—must also be specified. Like key size, the mode of operation can be specified as an optional argument in an additional clause `mode m` or declared as a part of the configuration. If left unspecified, the default mode of operation is used.

**Encryption and decryption** provide confidentiality, also known as secrecy, and are of the forms below.

\[
\text{encrypt text txt key } k
\]

Function `encrypt` encrypts text `txt` using key `k` and returns the resulting encrypted text. Function `decrypt` decrypts text `txt` using key `k` and returns the resulting decrypted text. They call the appropriate low-level library functions as determined by the key type, size, and mode.

**Signing and verification** provide authentication, integrity, and non-repudiation, and are of the forms below.

\[
\text{sign text txt key } k
\]
\[
\text{verify text txt sig } s \text{ key } k
\]

Function `sign` signs `txt` using key `k`. Signing has two modes: When the mode is detached, `sign` returns just the signature; when the mode is combined, `sign` returns the signed text. The mode of signing is determined by configuration, described below. The first form of function `verify`, for sign used in detached mode, verifies signature `s` against `txt` using key `k`, and returns true if verification succeeds and false otherwise. The second form of function `verify`, for sign used in combined mode, verifies signed `txt` using key `k` and returns the original text if verification succeeds and false otherwise.

**Declarative configuration.**

Configuration declaratively specifies values of parameters for cryptographic operations, and is of the form below, where configuration item `item` is assigned the value `value`.

\[
\text{configure item = value}
\]

Table 1 lists supported configuration items, their allowed values, and default values.

Declarative configuration allows security experts to exert control over the operation of high-level cryptographic abstractions in a clear and simple way. Proper default configuration values—one of capture the best practices as determined by security experts—are defined. This relieves developers of the burden of making choices about security algorithms, key sizes, modes, etc. for which they lack the relevant expertise or which are unnecessarily tedious to decide and code at a low level.

Configurations can be declared to apply globally, for particular sets of processes or communication channels, for particular scopes such as a method scope, or specified as optional arguments to individual operations. Configurations declared for an enclosed scope override those declared for an enclosing scope.

### 4 IMPLEMENTATION

We have developed a prototype implementation of SecAlgo as a Python module—a library of Python functions, one for each of the five basic operations described in the previous section.

SecAlgo is implemented on top of PyCrypto [49] for shared-key encryption using block ciphers (AES and Triple DES) in classical modes (CBC, CTR, CFB, OFB), message authentication code creation (HMAC), public key encryption (RSA), and public key digital signing (RSA, DSA). SecAlgo also provides support for Diffie-Hellman key pair generation using pre-established Diffie-Hellman parameters defined in [41, 46], but not yet abstraction for shared secret computation.

In addition, SecAlgo utilizes PyNaCl [72], PyCryptodome [36], and pyca/cryptography [71] (sometimes called cryptography.io [1]).
Table 1: Configuration items, allowed values, and the default value. * for key size indicates that allowed values depend on the
algorithm and backend library selected. ** for key size indicates the default value that follows the NIST advice [66].

<table>
<thead>
<tr>
<th>Item</th>
<th>Allowed values</th>
<th>Default value</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>key_type</td>
<td>shared, public</td>
<td>shared</td>
<td></td>
</tr>
<tr>
<td>key_type_shared</td>
<td>AES, Blowfish, 3DES, Salsa20, ChaCha20</td>
<td>AES</td>
<td>[63], [73], [67], [14], [13]</td>
</tr>
<tr>
<td>key_type_public</td>
<td>RSA, DSA, ECDSA</td>
<td>RSA</td>
<td>[56], [64]</td>
</tr>
<tr>
<td>key_size_shared</td>
<td>positive integer*</td>
<td>256*</td>
<td></td>
</tr>
<tr>
<td>key_size_public</td>
<td>positive integer*</td>
<td>2048**</td>
<td></td>
</tr>
<tr>
<td>block_cipher_mode</td>
<td>CBC, CTR, CFB, EAX, GCM, CCM, SIV, OCB</td>
<td>GCM</td>
<td>[33], [34], [39], [43]</td>
</tr>
<tr>
<td>sign_hash</td>
<td>SHA224, SHA256, SHA384, SHA512</td>
<td>SHA256</td>
<td>65</td>
</tr>
<tr>
<td>sign_mode</td>
<td>detached, combined</td>
<td>detached</td>
<td></td>
</tr>
<tr>
<td>backend_library</td>
<td>PyCrypto, PyNaCl, PyCryptodome, pyca/cryptography</td>
<td>PyCrypto</td>
<td>[49], [72], [36], [71]</td>
</tr>
</tbody>
</table>

for authenticated encryption modes for block ciphers (CCM [33], EAX [11], GCM [34], SIV [39], OCB [43]), safe stream ciphers (Salsa20 [14], ChaCha20 [13]), key derivation functions (HKDF [42]), elliptic curve digital signing [15], and elliptic curve Diffie-Hellman shared secret computation [12].

Some of these features (authenticated encryption modes, stream ciphers, elliptic curve digital signing) fall under our five abstractions. For the other operations, SecAlgo provides a high-level wrapper API around a fixed library implementation with a fixed set of parameters—choice of library implementation or configuration is currently not provided for these operations. Choice of RSA as the default for public key encryption and signing is due to current incomplete support of elliptic curve cryptography. This will be updated in future work.

Key generation.

Implementation of keygen uses functions in the low-level cryptographic libraries based on the type, size, and mode of operation specified.

To use only safe algorithms and implementations, SecAlgo checks all arguments of keygen against whitelists for approved combinations of algorithm types, key sizes, modes of operation, and hash functions. If the check fails, then keygen terminates and reports an error.

SecAlgo also makes sure that all key material—the bytes that form the shared secret for shared keys, or the modulus and exponents that form the public and private parts of the key pairs for RSA—is generated through calls to cryptographically strong pseudo-random number generators, which are usually provided by low-level cryptographic libraries and systems.

SecAlgo stores key material in a structure that also contains labels for algorithm type (value of key_type), key size (value of key_size_shared or key_size_public), block cipher mode of operation (value of block_cipher_mode), public vs. private part of the key pair if key_type is public, mode of signing (value of sign_mode), and the name of the hash function used by MAC and signing algorithms (value of sign_hash). The implementation uses Python’s dict to hold the key material and labels.

Encrypt, decrypt, sign, and verify.

For encryption, decryption, signing, and verification, SecAlgo manages tedious low-level details so that they remain hidden.

First, SecAlgo verifies that the input data is in a proper representation for submission to the cryptographic functions of the low-level cryptographic library. This includes making sure the input data is of the correct type and, if required by the encryption algorithm and mode of operation, of the proper size.

For example, PyCrypto requires that plaintexts are bytes-like objects [49] (a bytes-like object is one that supports the Buffer Protocol, such as bytes, bytesarray, or memoryview). If the input data is not of a compatible type and cannot be safely converted to the required type, then an error is signaled to the user.

Additionally, shared-key block algorithms have a block size—a fixed number of bytes to which the algorithm can be applied. Some modes of operation (such as CBC) require that every block of plaintext input must be of the block size. When using these modes for messages whose length is not divisible by the block size, SecAlgo automatically pads the data using a method that is safe for the encryption algorithm. For example, when using a block cipher in CBC mode, SecAlgo applies the PKCS7 [40] padding algorithm, which appends N bytes of value N to pad the plaintext to a multiple of the block size, where:

\[
N = \text{block size} - (\text{plaintext length} \mod \text{block size})
\]

Any padding is stripped from the decrypted data before it is returned. To avoid leaking information used by padding oracle attacks, SecAlgo does not report when padding errors cause decryption failure.

Also, SecAlgo generates and handles any auxiliary values used by the selected algorithm or mode of operation: initialization vectors, counters, and nonces. For example, Counter (CTR) mode encryption uses a counter, which can be any function that produces a sequence of block-size bytes values guaranteed not to repeat for a large number of iterations. SecAlgo follows a standard method [32] to generate the initial counter value by using a random value for the top-half of the counter and setting the bottom-half to 0. The random top-half value of the counter is prepended to the ciphertext.
For public key encryption, SecAlgo uses a straightforward hybrid encryption [25] scheme to encrypt arbitrary amounts of plaintext data. A single call to a public key encryption method can only encrypt a number of bytes that is less than the public key size. SecAlgo first checks the size of the plaintext to determine whether it can be encrypted directly using the public key algorithm, given the key size. If not, SecAlgo generates a 256-bit shared key and encrypts the data using AES in GCM mode. The new shared key is then decrypted with the public key algorithm. The public-key-encrypted AES key is prepended to the AES-in-GCM-mode-encrypted data, and this concatenation is returned.

The sign and verify functions are straightforward implementations of the behavior described in Section 3 using cryptographic functions specified by the configuration.

SecAlgo relies on the implementations in backend libraries to provide protection against side-channel attacks, such as timing channel mitigations built into some cryptographic libraries. Providing additional rigorous protection is future work.

5 APPLICATION: IMPLEMENTING SECURITY PROTOCOLS

To demonstrate the effectiveness of SecAlgo, we implemented a collection of well-known security protocols, such as Needham-Schroeder and others in the SPORE repository [77], as well as significant parts of more substantial protocols: TLS version 1.2 [31], Kerberos version 5 [61], and the Signal protocol [57, 68] (including significant parts of more substantial protocols: TLS version 1.2 [31], Kerberos version 5 [61], and the Signal protocol [57, 68]). Table 2 lists 10 of the protocols we implemented.

We implemented these protocols using SecAlgo plus the DistAlgo language [48, 50, 51], an extension of Python that provides high-level primitives for creating distributed processes, passing messages, and synchronization. The combination of SecAlgo with DistAlgo enables us to write clear, high-level implementations of security protocols.

The top part of Figure 1 shows the Denning-Sacco key distribution protocol [29] (a variation on the Needham-Schroeder public key authentication protocol [59]). There are three parties, an initiator (A), a responder (B), and a trusted authentication server (AS). The goal is to securely establish a new shared key (CK) known only by A and B. A message (m) is encrypted and sent on line 25, of the new shared key to transmit encrypted messages readable only by A and B. A message (m) is encrypted and sent on line 25, received on line 11, and decrypted on line 12.

Figure 2 shows the simplified Denning-Sacco key distribution protocol [19].


shared key and signature are decrypted, and then the signature on the shared key is verified. Lines 11-12 and 25 illustrate the use of the new shared key to transmit encrypted messages readable only by A and B. A message (m) is encrypted and sent on line 25, received on line 11, and decrypted on line 12.

```
from secalgo import *
configure(sign_mode = 'combined')

class RoleA (process):   # type A process
    def setup(skA, B, pkB): # take in params
        pass
    def run():
        k = keygen('shared') # new shared key
        send(((1, encrypt(sign(k, skA), pkB), to=B))
        await(some([received(((2, enc_m, from=_B))]
        m = decrypt(enc_m, k)
        output('Decrypted secret:', m)

class RoleB (process):   # type B process
    def setup(skB, pkA): # take in params
        self.m = 'secret' # set secret msg
    def run():
        await(False)

    def receive(msg=(1, enc_k, from=_A):
        k = verify(decrypt(enc_k, skB), pkA)
        if k: # k is not false
            send(((2, encrypt(n, k), to=A))

    def main():
        skA, pkA = keygen('public') # prv, pub key of A
        skB, pkB = keygen('public') # prv, pub key of B
        B = new(RoleB, (skB, pkA)) # create B
        A = new(RoleA, (skA, B, pkB)) # create A
        start(B)
        start(A)
```

Figure 2. Simplified Denning-Sacco key distribution protocol.
Each role in a protocol can be defined as a distinct process class. By extending process, a process in DistAlgo can send messages (lines 10 and 25), handle received messages (line 22), and await for synchronization conditions to become true (line 11).

This example illustrates several important features of SecAlgo:

1. Functions encrypt and decrypt can transparently provide both shared-key and public-key cryptographic operations as determined by the key type (shared-key on lines 12 and 25; public-key on lines 10 and 23).

2. Functions encrypt, decrypt, sign, and verify compose smoothly at a high level needing no extra effort (lines 10 and 23).

3. The return behavior of sign is controlled by the configuration statement on line 2, where sign_mode is set to combined. As a result, on line 10, sign returns a pair of k and the signature over k. On line 23, verify takes that pair as first argument, and so returns k itself, if verification succeeds (and False otherwise). On Line 24, we test the return from verify to ensure that the verification succeeded before using the key k to encrypt the secret in message 2.

If sign_mode had been set to detached on Line 2, the following changes to the program are required:

Line 10 is replaced with:

send((1, encrypt((k, sign(k, skA)), pkB)), to=B) where the key k is included separately in the body of message 1 because sign will return only the signature.

Lines 23-24 are replaced with:

k, sig = decrypt(enc_k, skB) if verify(k, sig, pkA):

where the key k and the signature sig must first be retrieved from the encrypted text before they can be passed to verify.

4. The developer is relieved of any extra tasks associated with using cryptographic operations. There is no need to generate an IV or a counter, or to pad plaintexts, for those algorithms that require it. All those tasks are managed in the background.

This last point demonstrates how SecAlgo simplifies decision-making about cryptographic operations. Even for the simplified Denning-Sacco protocol:

- The protocol contains three calls to keygen, two calls each to encrypt and decrypt, and one call each to sign and verify.
- Those three calls to keygen contain between 9 and 18 decisions regarding operation, algorithm, key size, mode of operation, padding, decisions with 66 possible outcomes.

These decision points are all occasions for a programmer, even an experienced one, to make mistakes. SecAlgo defaults ensure that all those decisions are made safely.

6 EXPERIMENTAL EVALUATION

We show that SecAlgo allows secure programs to be written much more easily than using lower-level libraries and incurs a minimum overhead. We also show in Appendix A that SecAlgo prevents seven main types of cryptographic misuse that are prevalent in mobile applications.

We compare measurements of programs that directly use SecAlgo with those that use the following lower-level libraries upon which SecAlgo is built:

- PyCrypto: The most widely-used general-purpose cryptography library for Python [1, Table 1].
- pyca/cryptography: The second-most widely-used general purpose cryptography library for Python [1, Table 1].
- PyCryptodome: A fork of PyCrypto extended to include newer cryptographic operations; still not in wide use [1, Table 1].
- PyNaCl: The best available Python interface for Curve 25519 elliptic curve cryptography [1].

The last three libraries provide additional operations not available in PyCrypto.

6.1 Code size and programming effort

SecAlgo has been used to implement over 20 security protocols, including those listed in Table 2. We compare implementations in SecAlgo with alternative implementations that use the lower-level cryptographic libraries PyCrypto, pyca/cryptography, PyCryptodome, and PyNaCl directly, and with abstract specifications written for protocol verification tools. We also compare with implementations in Java, C#, and Python for NS-SK, the corrected Needham–Schroeder shared key protocol.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS-SK</td>
<td>Corrected Needham-Schroeder protocol for key distribution by key server via shared key encryption [59]</td>
</tr>
<tr>
<td>NS-PK</td>
<td>Corrected Needham-Schroeder protocol for mutual authentication via public key encryption [59, 60]</td>
</tr>
<tr>
<td>DS</td>
<td>Denning-Sacco protocol for key distribution by key server and mutual authentication via public key encryption [29]</td>
</tr>
<tr>
<td>DS Simp</td>
<td>Simplified Denning-Sacco protocol for key distribution and mutual authentication via public key encryption [19]</td>
</tr>
<tr>
<td>DHKE-1</td>
<td>Diffie-Hellman key exchange protocol with mutual authentication via public key signatures [74]</td>
</tr>
<tr>
<td>SDH</td>
<td>Signed Diffie-Hellman key exchange protocol [23]</td>
</tr>
<tr>
<td>XSDH</td>
<td>Extended Triple Diffie-Hellman key exchange with mutual authentication via elliptic curve public key signatures [58]</td>
</tr>
<tr>
<td>DR</td>
<td>Double Ratchet (aka Axolotl) encrypted message exchange protocol via shared key authenticated encryption [79]</td>
</tr>
<tr>
<td>Signal</td>
<td>Signal: A ratching forward secrecy protocol for synchronous and asynchronous messaging environments [57, 68]</td>
</tr>
<tr>
<td>KRB-5</td>
<td>Kerberos, version 5, protocol for key distribution by key server and mutual authentication via shared key encryption [61]</td>
</tr>
<tr>
<td>TLS-1.2</td>
<td>Transport Layer Security (Handshake), version 1.2, for key exchange and mutual authentication via public key encryption [31]</td>
</tr>
</tbody>
</table>

Table 2: Well-known security protocols.
Table 3 gives the LOC (number of lines of code without comments) for the protocols listed in Table 2. We use LOC as an indirect measure of programming effort and program clarity. This is common practice in programming language literature.

### Ease of programming using SecAlgo.

The simplified function calls, automated generation of auxiliary values, declarative configuration, and carefully selected default options in the implementation of SecAlgo result in a reduction of the number of lines required to invoke cryptographic operations, and a simplification of those lines, when compared to other libraries.

For example, to encrypt a string \( pt \) using AES in CBC mode with a 32-byte key using PyCrypto, one must do the following:

```python
k = Random.new().read(32)
iv = Random.new().read(AES.blocksize)
cipher = AES.new(k, AES.MODE_CBC, iv)
ct = iv + cipher.encrypt(pad(pickle.dumps(pt)))
```

We can perform the same operation in SecAlgo as follows, where `key` is optional as in Python:

```python
k = keygen('shared')
crypt = crypt(pt, key = k)
```

We see a similar reduction in the number and complexity of lines of code for the other cryptographic operations supported by SecAlgo. In addition, we can alter the algorithm, keysize, and mode by declaring a new configuration, without having to alter either the call to `keygen` or the call to `encrypt`.

Our experience is that writing protocols using SecAlgo plus DistAlgo is much easier than using other languages and libraries. For simpler protocols like the first 6 in Table 2, we were able to implement them in SecAlgo plus DistAlgo, with LOC shown in column 2 of Table 3, by simply following their protocol narrations.

Seven relatively simple protocols, not listed in Table 2, were written by undergraduates and high-school students who, despite having had no or little familiarity with Python and being entirely new to SecAlgo and cryptography (as well as to DistAlgo and distributed programming), were able to complete the implementation in a couple of weeks with minimal assistance.

For X3DH, DR, and Signal protocols, we were able to easily use the core protocol specification from [79], which uses pseudocode that is simply Python code. We then added implementations of the lower-level cryptographic functions, which they call “external functions”, using other libraries for elliptic curve cryptography (Curve 25519 for both signing and Diffie-Hellman) and key derivation (HKDF).

### Comparison with using PyCrypto.

We also implemented the protocols in Table 2 using PyCrypto (and PyNaCl for X3DH, DR, and Signal) plus DistAlgo. Column 3 of Table 3 shows the LOC of these implementations. Executing these programs produces the same calls to the underlying cryptography libraries as those in column 2.

Table 4 lists the number of calls to SecAlgo functions that appear in each protocol implementation. Each SecAlgo function call uses 1 or more fewer lines of code compared with using PyCrypto and other lower-level libraries.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>keygen</th>
<th>encrypt</th>
<th>decrypt</th>
<th>sign</th>
<th>verify</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS-SK</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>NS-PK</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>DS</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>DS Simp</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>DHKE-1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>SDH</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>X3DH</td>
<td>18</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>DR</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>Signal</td>
<td>29</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>41</td>
</tr>
<tr>
<td>KRB-5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>TLS-1.2</td>
<td>14</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 4: Number of calls to SecAlgo functions in each protocol implementation.
Comparison with abstract protocol specifications.

Columns 4 to 8 of Table 3 show LOC of abstract specifications for the protocols in Table 2 in the best security protocol specification languages (for all we could find), as written by experts in these languages. These specifications are not executable, and are used as input to specialized verifiers of the respective languages. Our executable SecAlgo programs are actually similar in size to the most abstract of these specifications, as evidenced by the similar LOC. The SecAlgo plus DistAlgo implementations of all but the last 2 protocols have smaller LOC than all of the abstract specifications except for those in Scyther (SPDL [28]).

For the last 2 protocols, TLS and Kerberos, the most significant cause of the larger LOC of the SecAlgo plus DistAlgo implementation is functionalities omitted from the abstract specifications. For example, the abstract specifications of TLS include only the TLS Handshake protocol, whereas the SecAlgo plus DistAlgo implementation also includes the TLS Record protocol and the TLS ChangeCipherSpec protocol. For Kerberos, none of the abstract specifications construct tickets with actual timestamps, or use those timestamps to validate tickets, whereas the SecAlgo plus DistAlgo implementation does both.

Comparison with using other programming languages for NS-SK.

Table 5 compares implementations of NS-SK in C#, Java, PyCrypto plus Python, SecAlgo plus Python, PyCrypto plus DistAlgo and SecAlgo plus DistAlgo. The implementations use different libraries for distributed programming and cryptographic operations. These implementations were developed by ourselves or with our supervision, and they represent our best effort, so far, to use each language in the best way. For LOC comparison, we formatted the programs according to the suggested style of each language.

<table>
<thead>
<tr>
<th>Language</th>
<th>Crypto library</th>
<th>NS-SK</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>.NET Cryptography [55]</td>
<td>364</td>
</tr>
<tr>
<td>Java</td>
<td>JCA [69]</td>
<td>351</td>
</tr>
<tr>
<td>Python</td>
<td>PyCrypto [49]</td>
<td>217</td>
</tr>
<tr>
<td>Python</td>
<td>SecAlgo</td>
<td>170</td>
</tr>
<tr>
<td>DistAlgo</td>
<td>PyCrypto [49]</td>
<td>68</td>
</tr>
<tr>
<td>DistAlgo</td>
<td>SecAlgo</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 5: LOC of NS-SK implementations using different languages and libraries.

The C# and Java programs required much more effort than the Python programs, which required much more effort than the DistAlgo programs. This is also evident in the LOC comparison. Our experience writing these implementations confirmed that using high-level cryptographic and communication abstractions significantly help reduce program size and programming effort and increase program clarity.

6.2 Running times and overhead

We discuss three running time experiments measuring (1) the time taken by SecAlgo functions compared with using the underlying lower-level cryptographic library directly, (2) the time of cryptographic operations vs. message passing in protocols, and (3) the time of NS-SK implementations using SecAlgo vs. using PyCrypto on top of DistAlgo and Python, vs. implementations in Java and C#.

All reported running times are CPU times measured on an Intel Core i5-5250U processor of 2.70GHZ with 16GB of DDR3L memory, running Ubuntu 17.10, DistAlgo 1.0.12, and Python 3.6.3. PyCrypto 2.6, PyCryptodome 3.5.1, pyca/cryptography 2.2.2, and PyNaCl 1.2.1 are used for cryptographic operations. For all experiments the Python garbage collector was disabled. For each measurement, protocols and cryptographic operations are run in a loop for at least one second and the CPU time is averaged over the number of iterations in order to get an accurate estimate of the CPU time for a single execution of that protocol or operation. Each of those measurements is repeated at least 50 times, and the average is taken.

Overhead of SecAlgo abstractions.

We fix a configuration—an algorithm, a key size, a mode of operation, and a padding—and measure the running time of each SecAlgo primitive. We measure the same operation written using PyCrypto directly. The measurement for using PyCrypto directly also includes the time needed to encode input to cryptographic functions as byte strings and the reverse for output from the cryptographic functions. This is done because it is required by the low-level libraries. We use the pickle library for Python for encoding and decoding.

Table 6 shows that SecAlgo primitives impose small overhead for all cryptographic primitives compared with using PyCrypto. The overhead is ≤ 4.5 microseconds for all shared key primitives, The overhead is ≤ 13.02 microseconds for all public key primitives except for 340.54 microseconds for keygen, but all are < 1%.

Public keys used by SecAlgo are 8 times as large as those used for shared key cryptography (2048-bit vs. 256-bit). This means that public key primitives may have more varied increases in running times due to memory effect, as observed. At the same time, because public key primitives are much more expensive, the percentage increases may be much smaller, again as observed.

Running time of cryptographic operations in total protocol time.

To understand the running times of cryptographic operations among total protocol time, we measure these times for each protocol in Table 2, and we show the contributing factors by counting the number of calls to different cryptographic functions and the number of messages passed.

For protocol time, we measure the time used by each role excluding process setup time, and sum over all roles. For the time of all cryptographic operations, which we call library time, we measure the time of each SecAlgo function call and sum over all calls. We collect the counts of calls and messages for the measured execution of each protocol.

Table 7 shows the results, grouped by the kinds of cryptographic functions called and sorted by decreasing library time in each
Table 6: Cryptographic operations and configurations used, CPU times (in microseconds) when using PyCrypto and using SecAlgo, and time increase (in microseconds) and percentage increase from PyCrypto time to SecAlgo time.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Configuration</th>
<th>PyCrypto</th>
<th>SecAlgo</th>
<th>Increase</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared key</td>
<td>keygen</td>
<td>AES, 256, CBC, PKCS7</td>
<td>47.36</td>
<td>49.55</td>
<td>2.19</td>
</tr>
<tr>
<td>encrypt</td>
<td>AES, 256, CBC, PKCS7</td>
<td>65.93</td>
<td>70.43</td>
<td>4.50</td>
<td>6.83</td>
</tr>
<tr>
<td>decrypt</td>
<td>AES, 256, CBC, PKCS7</td>
<td>14.46</td>
<td>16.71</td>
<td>2.25</td>
<td>15.56</td>
</tr>
<tr>
<td>sign</td>
<td>HMAC, 256, SHA512</td>
<td>16.9</td>
<td>18.65</td>
<td>1.75</td>
<td>10.36</td>
</tr>
<tr>
<td>verify</td>
<td>HMAC, 256, SHA512</td>
<td>17.26</td>
<td>18.84</td>
<td>1.58</td>
<td>9.15</td>
</tr>
<tr>
<td>Public key</td>
<td>keygen</td>
<td>RSA, 2048</td>
<td>124,526.75</td>
<td>124,867.29</td>
<td>340.54</td>
</tr>
<tr>
<td>encrypt</td>
<td>RSA, 2048, OAEP</td>
<td>1,322.00</td>
<td>1,322.42</td>
<td>0.42</td>
<td>0.03</td>
</tr>
<tr>
<td>decrypt</td>
<td>RSA, 2048, OAEP</td>
<td>3100.17</td>
<td>3106.41</td>
<td>6.24</td>
<td>0.20</td>
</tr>
<tr>
<td>sign</td>
<td>RSA, 2048, PKCS1</td>
<td>2995.29</td>
<td>3008.31</td>
<td>13.02</td>
<td>0.43</td>
</tr>
<tr>
<td>verify</td>
<td>RSA, 2048, PKCS1</td>
<td>698.92</td>
<td>705.21</td>
<td>6.29</td>
<td>0.90</td>
</tr>
</tbody>
</table>

Comparison with using Python and PyCrypto on NK-SK.

Figure 3 shows the running times of NS-SK written using SecAlgo and PyCrypto on top of DistAlgo and Python, for all 4 combinations, measured by repeating NS-SK on increasing numbers of runs.

All 4 implementations show a linear increase in running time as the number of runs increases. The difference between using SecAlgo and using PyCrypto, on top of DistAlgo or on top of Python, is small: at most 2.4 seconds and between -2% and 16%. The difference can sometimes be negative because of the small overhead of SecAlgo and the usual variation in running times of multi-process protocols even when averaged over 50 runs.

Figure 3: Running times of NS-SK on increasing numbers of protocol runs.

Using DistAlgo is about 5 times as slow as Python, but that is expected and is the subject of DistAlgo compilation and optimizations studied separately [50].

The main result is that whether using DistAlgo or Python, using SecAlgo is at most a small increase over using PyCrypto directly, while being safe and much simpler to use.

7 RELATED WORK AND CONCLUSION

There have been many efforts at building better cryptographic libraries providing simpler interfaces. These include the NaCl library [16, 17] for C and C++; libsodium [38], a portable version of NaCl with a slightly improved interface; the pycryptography library [71] for Python; the Charm library [2, 3] for Python; the Keyczar library [30, 78] for C++, Java, and Python; and the Tink library [5] for C++, Java, Go, and Objective-C.

These libraries simplify use of cryptographic operations in ways similar to SecAlgo. Simplifying techniques to reduce the number of decisions for users include: (1) requiring fewer inputs from the user, (2) handling tedious, routine tasks automatically behind the
Table 7: Number of calls to different cryptographic functions (with expensive ones first), CPU times (in milliseconds) of protocol run and library calls, difference between the two times, and number of messages passed. An empty entry denotes 0. * for TLS1.2 indicates that the number of messages can differ when different branching conditions hold; our experiments used the condition for which 9 messages are passed.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>pow key-gen</th>
<th>RSA key-gen</th>
<th>pow sign veri.</th>
<th>pow rest</th>
<th>Library time</th>
<th>Protocol time</th>
<th>Time diff.</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDH</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>160.08</td>
<td>161.71</td>
<td>1.63</td>
<td>3</td>
</tr>
<tr>
<td>DHKE-1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>42.60</td>
<td>45.89</td>
<td>3.29</td>
<td>3</td>
</tr>
<tr>
<td>NS-PK</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td>19.87</td>
<td>27.87</td>
<td>7.99</td>
<td>7</td>
</tr>
<tr>
<td>DS</td>
<td>4</td>
<td>6</td>
<td></td>
<td>1</td>
<td>17.59</td>
<td>22.76</td>
<td>5.17</td>
<td>3</td>
</tr>
<tr>
<td>DS Simp</td>
<td>2</td>
<td>2</td>
<td></td>
<td>3</td>
<td>8.34</td>
<td>10.40</td>
<td>2.06</td>
<td>2</td>
</tr>
<tr>
<td>TLS 1.2</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>33</td>
<td>20.33</td>
<td>13.43</td>
<td>9-13*</td>
</tr>
<tr>
<td>Signal</td>
<td>11</td>
<td>19</td>
<td>29</td>
<td></td>
<td>8.40</td>
<td>19.05</td>
<td>10.65</td>
<td>8</td>
</tr>
<tr>
<td>DR</td>
<td>6</td>
<td>9</td>
<td>25</td>
<td></td>
<td>4.93</td>
<td>9.70</td>
<td>4.77</td>
<td>6</td>
</tr>
<tr>
<td>X3DH</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td></td>
<td>3.72</td>
<td>8.42</td>
<td>4.70</td>
<td>4</td>
</tr>
<tr>
<td>KRbV5</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td>0.75</td>
<td>8.45</td>
<td>7.70</td>
<td>6</td>
</tr>
<tr>
<td>NS-SK</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>0.61</td>
<td>11.61</td>
<td>10.99</td>
<td>7</td>
</tr>
</tbody>
</table>

Acknowledgments

We thank Rahul Sihag for help in detailed running time measurements and analysis, and Yuege Chen and Wenjun Qu for protocol implementations in C# and Java. This work was supported in part by NSF under grants CCF-1414078 and CNS-1421893 and ONR under grant N000141512208.
Table 8: Misuse type and number of apps containing that type, plus total number of apps studied, using the misuse analysis systems CryptoLint [35], CMA [75], CNKX [24], and CDRep [52]. Misuse types prevented by SecAlgo are listed; three other types studied [24, 35, 52] (a constant salt for password-based encryption (PBE), < 1000 iterations for PBE, and improper seeding for Java SecureRandom objects), not prevented by SecAlgo, are not listed. '-' means that the study did not report about instances of the corresponding misuse type.

<table>
<thead>
<tr>
<th>Misuse type</th>
<th>Description</th>
<th>Reported number of apps with this misuse type by study</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CryptoLint</td>
</tr>
<tr>
<td>M1K</td>
<td>Insufficient key size</td>
<td>-</td>
</tr>
<tr>
<td>M2K</td>
<td>Constant or hardcoded keys</td>
<td>3644</td>
</tr>
<tr>
<td>M1S</td>
<td>Encryption in ECB mode</td>
<td>7656</td>
</tr>
<tr>
<td>M2S</td>
<td>Encryption with predictable IV</td>
<td>1932</td>
</tr>
<tr>
<td>M3S</td>
<td>Encryption with obsolete algorithm</td>
<td>-</td>
</tr>
<tr>
<td>M1A</td>
<td>RSA encryption without OAEP</td>
<td>-</td>
</tr>
<tr>
<td>M1H</td>
<td>Hashing with obsolete algorithm</td>
<td>-</td>
</tr>
<tr>
<td>Sum</td>
<td>Sum of numbers above</td>
<td>13232</td>
</tr>
</tbody>
</table>

Table 8: Misuse type and number of apps containing that type, plus total number of apps studied, using the misuse analysis systems CryptoLint [35], CMA [75], CNKX [24], and CDRep [52]. Misuse types prevented by SecAlgo are listed; three other types studied [24, 35, 52] (a constant salt for password-based encryption (PBE), < 1000 iterations for PBE, and improper seeding for Java SecureRandom objects), not prevented by SecAlgo, are not listed. '-' means that the study did not report about instances of the corresponding misuse type.

<table>
<thead>
<tr>
<th>Misuse type</th>
<th>Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1K</td>
<td>excluded from whitelist of approved key sizes, safe defaults</td>
</tr>
<tr>
<td>M2K</td>
<td>keygen generates random key at runtime</td>
</tr>
<tr>
<td>M1S</td>
<td>excluded from whitelist of approved block modes of operation</td>
</tr>
<tr>
<td>M2S</td>
<td>encrypt generates random IV when needed</td>
</tr>
<tr>
<td>M3S</td>
<td>excluded from whitelists of approved encryption algorithms</td>
</tr>
<tr>
<td>M1A</td>
<td>encrypt uses OAEP padding for RSA encryption, no alternative</td>
</tr>
<tr>
<td>M1H</td>
<td>excluded from whitelist of approved hashing algorithms</td>
</tr>
</tbody>
</table>

Table 9: Summary of the way in which each misuse type is prevented by SecAlgo.

Table 9: Summary of the way in which each misuse type is prevented by SecAlgo.

- **M1K: Insufficient key size.** This issue is handled by the keygen abstraction. The default key sizes for all algorithms are safe as they guarantee at least 112 bits of security, which NIST has determined as the minimum security strength allowable until 2030 [66]. A key size given explicitly at a call to keygen is checked against a whitelist for the algorithm and if the key size is found to be insufficient, SecAlgo throws an exception.
- **M2K: Hard-coded keys.** Hard-coded keys are unsafe because they can be extracted by binary disassembly. SecAlgo inhibits the use of hard-coded keys through easy generation of keys using keygen and, in a planned extension, easy secure storage of keys.
- **M1S: Encryption in ECB mode.** Creation of a key for ECB mode is prevented during keygen because ECB is not included in the the whitelist of allowed block cipher modes in SecAlgo. The whitelist is checked again when the key is used preventing the use of keys whose tags have been manually altered in an attempt to encrypt with ECB mode. Any attempt to use unsafe block modes like ECB, detected by checking the whitelist of approved modes will be reported as an error at runtime.
- **M2S: Encryption with predictable IV.** The default behaviour of encrypt generates IVs automatically, thus preventing predictable IVs. SecAlgo uses a cryptographically strong random number generator to generate the random data block to use as the IV as directed by NIST SP 800-38A [32].
- **M3S: Encryption with obsolete algorithm.** As for M1S, this misuse is prevented by having a whitelist of safe algorithms. Obsolete algorithms like DES, ARC2 and ARC4 stream ciphers are not allowed by SecAlgo abstractions keygen, encrypt, and decrypt. Any attempt to use an obsolete algorithm will be reported as an error at runtime.
- **M1A: RSA encryption without OAEP.** Optimal Asymmetric Encryption Padding (OAEP) is the default padding scheme used with RSA by SecAlgo. SecAlgo does not offer any alternative to OAEP and thereby ensures safety.
- **M1H: Hashing with obsolete algorithm.** Unsafe hashing algorithms like MD2, MD4, MDS and SHA-1 are not in the whitelist of allowed hashing algorithms in SecAlgo. Any attempt to use an obsolete algorithm will be reported as an error at runtime, as for M1S.

REFERENCES

