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.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Cryptography*; • **Software and its engineering → Very high level languages**;

KEYWORDS

cryptographic API, declarative configuration, high-level abstraction

ACM Reference Format:
Christopher Kane, Bo Lin, Saksham Chand, Scott D. Stoller, and Yanhong A. Liu. 2019. High-Level Cryptographic Abstractions . In *14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3338504.3357343

PLAS'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6836-0/19/11…$15.00
https://doi.org/10.1145/3338504.3357343

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 pyca/cryptography'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`.

```
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.

```
encrypt text txt key k
decrypt 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.

```
sign text txt key k
verify text txt sig s key k
verify text txt 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 $value$.

```
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—ones that 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])

Item	Allowed values	Default value	References
key_type	shared, public	shared	
key_type_shared	AES, Blowfish, 3DES, Salsa20, ChaCha20	AES	AES [63], Blowfish [73], 3DES [67], Salsa20 [14], ChaCha20 [13]
key_type_public	RSA, DSA, ECDSA	RSA	RSA [56], DSA, ECDSA [64]
key_size_shared	positive integer*	256**	
key_size_public	positive integer*	2048**	
block_cipher_mode	CBC, CTR, CFB, EAX, GCM, CCM, SIV, OCB	GCM	CBC, CTR, CFB [32], EAX [11], GCM [34], CCM [33], SIV [39], OCB [43]
sign_hash	SHA224, SHA256, SHA384, SHA512	SHA256	[65]
sign_mode	detached, combined	detached	
backend_library	PyCrypto, PyNaCl, PyCryptodome, pyca/cryptography	PyCrypto	PyCrypto [49], PyNaCl [72], PyCryptodome [36], pyca/cryptography [71]

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].

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 operating 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`, `bytearray`, or `memoryview`). If the input data is not of a compatible type and can not 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:

$$N = block\_size - (plaintext\_length \ \% \ block\_size)$$

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 encrypted 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 its components the Double Ratchet protocol [79] and the Extended Triple Diffie-Hellman (X3DH) protocol [58]). 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 to A and B. A acquires certificates containing its own public key (CA) and B's public key (CB). These certificates are passed by A to B in message 3. With the keys contained in these certificates, A and B use public key cryptography to protect the confidentiality and integrity of the new shared key, and to authenticate each other.

A simplified version of the Denning-Sacco protocol [19] is shown in the bottom part of Figure 1. It assumes that A and B both already possess the other's public key. As a result, A does not need to get certificates containing those public keys from AS, which eliminates AS and the first two messages. The simplified protocol also does away with the timestamp (T) associated with the new shared key. The simplified protocol extends the original protocol to include sending an encrypted application message.

Figure 2 shows the simplified Denning-Sacco protocol in SecAlgo plus DistAlgo. On line 10, message 1 of the simplified protocol is sent, and on lines 23-24 that message is received, the encrypted

(1) $A \rightarrow AS : A, B$
(2) $AS \rightarrow A : CA, CB$
(3) $A \rightarrow B : CA, CB, \{\{CK, T\}_{S_A}\}_{P_B}$

Where:

- A and B are users, and AS is a centralized key distribution facility called an Authentication Server.
- T is a timestamp
- P_X and S_X denote user X's public key and secret (signature) key respectively
- $CA = \{A, P_A, T\}_{S_{AS}}$ and $CB = \{B, P_B, T\}_{S_{AS}}$
- The key CK is then used for encrypting messages transmitted between A and B.

---

(1) $A \rightarrow B : \{\{CK\}_{S_A}\}_{P_B}$
(2) $B \rightarrow A : \{msg\}_{CK}$

Figure 1: Top: Denning-Sacco public key distribution protocol [29, p. 535]. Bottom: 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.

```
1  from secalgo import *
2  configure(sign_mode = 'combined')
3
4  class RoleA (process):        # type A process
5    def setup(skA, B, pkB):     # take in params
6      pass
7
8    def run():
9      k = keygen('shared')      # new shared key
10     send((1, encrypt(sign(k, skA), pkB)), to=B)
11     await(some(received((2, enc_m), from_=_B)))
12     m = decrypt(enc_m, k)
13     output('Decrypted secret:', m)
14
15 class RoleB (process):        # type B process
16   def setup(skB, pkA):        # take in params
17     self.m = 'secret'         # set secret msg
18
19   def run():
20     await(False)
21
22   def receive(msg=(1, enc_k), from_=A):
23     k = verify(decrypt(enc_k, skB), pkA)
24     if k:                     # k is not false
25       send((2, encrypt(m, k)), to=A)
26
27 def main():
28   skA,pkA = keygen('public') # prv,pub key of A
29   skB,pkB = keygen('public') # prv,pub key of B
30   B = new(RoleB, (skB, pkA)) # create B
31   A = new(RoleA, (skA,B,pkB))# create A
32   start(B)
33   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.

Protocol	Description
NS-SK	Corrected Needham-Schroeder protocol for key distribution by key server via shared key encryption [59]
NS-PK	Corrected Needham-Schroeder protocol for mutual authentication via public key encryption [59, 60]
DS	Denning-Sacco protocol for key distribution by key server and mutual authentication via public key encryption [29]
DS Simp	Simplified Denning-Sacco protocol for key distribution and mutual authentication via public key encryption [19]
DHKE-1	Diffie-Hellman key exchange protocol with mutual authentication via public key signatures [74]
SDH	Signed Diffie-Hellman key exchange protocol [23]
X3DH	Extended Triple Diffie-Hellman key exchange with mutual authentication via elliptic curve public key signatures [58]
DR	Double Ratchet (aka Axolotl) encrypted message exchange protocol via shared key authenticated encryption [79]
Signal	Signal: A ratcheting forward secrecy protocol for synchronous and asynchronous messaging environments [57, 68]
KRB-5	Kerberos, version 5, protocol for key distribution by key server and mutual authentication via shared key encryption [61]
TLS-1.2	Transport Layer Security (Handshake), version 1.2, for key exchange and mutual authentication via public key encryption [31]

Table 2: Well-known security protocols.

Protocols	SecAlgo+DistAlgo	PyCrypto+DistAlgo	Scyther	AVISPA	ProVerif	Tamarin	CryptoVerif
NS-PK	47	96	36	55	107	109	116
NS-SK	46	68	41		82		94
DS	50	102			96		120
DS Simp	26	69					
DHKE-1	63	113	41				
SDH	39	73	35		41	48	89
X3DH	140	151					
DR	182	199					
Signal	321	349					
KRB-5	171	213	94	137			186
TLS	(v. 1.2) 430	(v. 1.2) 478	(v. 1.0) 53	(v. 1.0) 107	(v. 1.3) 397	(v. 1.0) 128	

Table 3: LOC of protocol implementations (executable on distributed machines) in SecAlgo+Distalgo and PyCrypto+DistAlgo, and of abstract specifications in the languages and tools: Scyther [27], AVISPA [8], ProVerif [21], Tamarin [54], and CryptoVerif CryptoVerifSource. Our implementations of X3DH, DR, and Signal include 58 lines of Python code taken directly from the specification [79]. Empty entry means we did not find a corresponding specification.

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:

```
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:

```
k = keygen('shared')
ct = encrypt(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. As a result, protocol implementations written using SecAlgo are shorter and simpler than those written using PyCrypto and other lower-level libraries.

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

Table 4: Number of calls to SecAlgo functions in each protocol implementation.

The average percentage difference in LOC across all implementations written using SecAlgo (shown in column 2 of Table 3) compared to those written using low-level libraries (shown in column 3 of Table 3) is 31%, that is, using SecAlgo reduces LOC of protocol implementations by almost a third on average.

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.

Language	Crypto library	NS-SK
C#	.NET Cryptography [55]	364
Java	JCA [69]	351
Python	PyCrypto [49]	217
Python	SecAlgo	170
DistAlgo	PyCrypto [49]	68
DistAlgo	SecAlgo	46

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, PyCryptdome 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 cryptographics functions called and sorted by decreasing library time in each

	Operation	Configuration	PyCrypto	SecAlgo	Increase	% Increase
Shared key	keygen	AES, 256, CBC, PKCS7	47.36	49.55	2.19	4.62
	encrypt	AES, 256, CBC, PKCS7	65.93	70.43	4.50	6.83
	decrypt	AES, 256, CBC, PKCS7	14.46	16.71	2.25	15.56
	sign	HMAC, 256, SHA512	16.9	18.65	1.75	10.36
	verify	HMAC, 256, SHA512	17.26	18.84	1.58	9.15
Public key	keygen	RSA, 2048	124,526.75	124.867.29	340.54	0.27
	encrypt	RSA, 2048, OAEP	1,322.00	1,322.42	0.42	0.03
	decrypt	RSA, 2048, OAEP	3100.17	3106.41	6.24	0.20
	sign	RSA, 2048, PKCS1	2995.29	3008.31	13.02	0.43
	verify	RSA, 2048, PKCS1	698.92	705.21	6.29	0.90

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.

group. Cryptogrpahic functions are listed in the order of expensive ones first: modular exponentiation (pow) for Diffie-Hellman, RSA functions, elliptic curve (EC) functions, and shared key functions (SK); among RSA functions, keygen, decrypt and sign that use private keys, and encrypt and verify that use public keys; among EC functions, keygen and the rest.

For library time, we see that it is almost fully determined by the counts of calls to more expensive functions, with two exceptions: (1) SDH and DHKE-1 both have the same numbers of expensive calls, especially power function pow to compute Diffie-Hellman shared secrets, but the larger time for SDH is because it uses values that are 3 times as large; (2) Signal uses EC, but it has many more calls to EC keygen and thus a slightly larger library time than DS Simp and TLS 1.2 that use RSA but have few calls of non-keygen functions. In fact, with the exception of Signal, the library time is sorted completely in decreasing order.

Protocol time is also mostly in decreasing order, but with three exceptions: TLS 1.2, Signal, and NS-SK. This is because protocol time is also affected by the number of messages passed during the protocol run. In fact, the three exceptions are from protocols that have the most messages.

We consider the difference between protocol and library times. We see that for each group, a larger time difference corresponds to a larger number of messages, with one exception: SDH and DHKE-1 both have 3 messages, but the larger difference for DHKE-1 is due to additional local, non-cryptographic computations in DHKE-1 but not in SDH.

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.

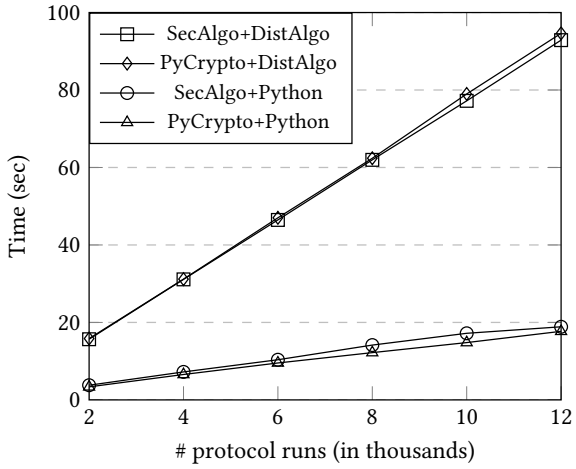

Figure 3: Running times of NS-SK on increasing numbers of protocol runs.

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 pyca/cryptography 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

Protocol	pow	RSA			EC		SK	Library time	Protocol time	Time diff.	Messages
		key-gen	dec. sign	enc. veri.	key-gen	rest					
SDH	2	2	2	2				160.08	161.71	1.63	3
DHKE-1	2	2	2	4				42.60	45.89	3.29	3
NS-PK			5	5				19.87	27.87	7.99	7
DS			4	6			1	17.59	22.76	5.17	3
DS Simp			2	2			3	8.34	10.40	2.06	2
TLS 1.2			1	3			33	6.90	20.33	13.43	9-13*
Signal					11	19	29	8.40	19.05	10.65	8
DR					6	9	25	4.93	9.70	4.77	6
X3DH					5	10	4	3.72	8.42	4.70	4
KRBv5							26	0.75	8.45	7.70	6
NS-SK							11	0.61	11.61	10.99	7

Table 7: Number of calls to diferent 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.

scenes, (3) supporting better default configurations, and (4) removing unsafe algorithms and implementations.

However, these libraries fall short when compared with SecAlgo. Charm provides simplified use for only a single operation—shared key authenticated encryption. pyca/cryptography provides simplified use of only shared key authenticated encryption and X.509 certificate handling. Keyczar and Tink provide only shared key authenticated encryption, hybrid encryption, signing, and message authentication code creation. NaCl and libsodium provide a much more complete set of cryptographic operations, but provide little to no configurability but only one algorithm for most cryptographic operations.

Acar et al. [1] study the usability of five Python cryptographic libraries: PyCrypto [49], M2Crypto [76], Keyczar [30, 78], pyca/cryptography [71], and PyNACL [72] (a Python binding of NACL). They found that clear documentation and concrete code examples were the most significant factors determining whether subjects produced solutions that work. Furthermore, they found that code written with simplified APIs were much more likely to be secure, while code written with low-level libraries were more likely to contain mistakes that compromised their security. SecAlgo provides higher-level, simpler APIs than these previous libraries.

Egele et al. [35, p. 81] study cryptographic misuse in Android and propose mitigation strategies: (1) introduction of better default configurations in cryptographic libraries and (2) provision of better, more complete documentation of cryptographic libraries. SecAlgo realizes the first by default configurations that implement best security practice and allows the second to be made much simpler and easier to use.

FixDroid [62] is an IDE plug-in for the Android SDK that identifies cryptographic mistakes in source code, as it is written, and provides suggested corrections. CogniCrypt [44] automatically generates Java code for a collection of common cryptographic tasks (e.g., encrypting data with a password, storing passwords, secure communication, etc.) and performs static analysis to verify that generated code is properly integrated into the user's application. CDRep [52] acts directly on Android binaries by using static analysis to detect cryptographic misuses and then generates and applies patches to correct them. Use of SecAlgo allows many tasks of such tools to be greatly simplified or completely eliminated.

Security protocol specification languages are for abstract formulation and verification of security protocols. Scyther [26, 27], AVISPA [6, 7], ProVerif [20, 21], and CryptoVerif [18, 22] are process or role oriented similar to SecAlgo plus DistAlgo. Tamarin [53, 54] models the state of the protocol as a multi-set of facts and models protocol actions as rewrite rules operating on these facts. SecAlgo plus DistAlgo programs are simpler than even abstract specifications written in most of these formal specification languages. Unlike these formal specification languages, SecAlgo is for building actual implementations of security protocols as well as full-fledged secure applications.

In conclusion, SecAlgo provides simpler and more powerful high-level abstractions for cryptographic operations and allows security protocols and applications to be written more easily and clearly. Future work includes possible further optimization of the implementation to minimize performance overhead, extension to support more combinations of best cryptographic functions from different libraries, static checking and optimization of these combinations, more extensive use and evaluation of the abstractions, and translation into languages of protocol verification tools such as ProVerif and Scyther for formal verification.

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.

A MISUSE PREVENTION

To validate SecAlgo against main types of cryptographic misuse, we surveyed studies of misuses that occur in mobile applications.

Misuse type	Description	Reported number of apps with this misuse type by study			
		CryptoLint	CMA	CNKX	CDRep
M1K	Insufficient key size	-	1	7	-
M2K	Constant or hardcoded keys	3644	0	4	882
M1S	Encryption in ECB mode	7656	7	16	887
M2S	Encryption with predictable IV	1932	8	2	979
M3S	Encryption with obsolete algorithm	-	8	16	-
M1A	RSA encryption without OAEP	-	3	2	-
M1H	Hashing with obsolete algorithm	-	38	16	5582
Sum	Sum of numbers above	13232	65	63	8330
Total apps	Total number of apps analyzed	11748	45	49	8640

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.

Misuse type	Prevention
M1K	excluded from whitelist of approved key sizes, safe defaults
M2K	`keygen` generates random key at runtime
M1S	excluded from whitelist of approved block modes of operation
M2S	`encrypt` generates random IV when needed
M3S	excluded from whitelists of approved encryption algorithms
M1A	`encrypt` uses OAEP padding for RSA encryption, no alternative
M1H	excluded from whitelist of approved hasing algorithms

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

Table 8 presents results of our evaluation using four such studies. It shows that SecAlgo prevents seven main types of cryptographic misuse out of a total of ten. Note that each misuse type reported occurs at least once in each of the applications counted. Thus even the sum from CryptoLint alone means that SecAlgo abstractions prevent at least 13232 instances of types of cryptographic misuse.

We describe how exactly SecAlgo prevents all of the misuse types listed in Table 8. They are summarized in Table 9.

- **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, MD5 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

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. 154–171. https://doi.org/10.1109/SP.2017.52

[2] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. 2013. Charm: a framework for

rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering* 3, 2 (2013), 111–128. https://doi.org/10.1007/s13389-013-0057-3

[3] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. 2015. Charm: A Framework for Rapidly Prototyping Cryptosystems. https://github.com/JHUISI/charm. Last accessed: July 22, 2019.

[4] Ross Anderson. 1993. Why Cryptosystems Fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS '93)*. 215–227. https://doi.org/10.1145/168588.168615

[5] Haris Andrianakis, Daniel Bleichenbacher, Thai Duong, Thomas Holenstein, Charles Lee, Quan Nguyen, Bartosz Przydatek, and Veronika Slivova. 2018. Tink. https://github.com/google/tink. Latest release: 1.2.1. Last accessed: November 16, 2018.

[6] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. 2005. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*. Springer, 281–285. https://doi.org/10.1007/11513988_27

[7] Alessandro Armando, Michael Rusinowitch, David Basin, and David Von Oheimb. 2006. AVISPA: Automated Validation of Internet Security Protocols and Applications. http://www.avispa-project.org/. Latest release: 1.1. Last accessed: July 31, 2016.

[8] Alessandro Armando, Michael Rusinowitch, David Basin, and David Von Oheimb. 2006. AVISPA Library: A collection of security protocol specifications written in HLPSL. http://avispa-project.org/library/avispa-library.html. Last accessed: November 8, 2016.

[9] Rebecca Balebako, Abigail Marsh, Jialiu Lin, Jason I. Hong, and Lorrie Fath Cranor. 2014. The Privacy and Security Behaviors of Smartphone App Developers. In *Proceedings of the 2014 Workshop on Usable Security (USEC 2014)*. Internet Society.

[10] Mihir Bellare, Roch Guérin, and Phillip Rogaway. 1995. XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '95)*. Springer, 15–28. https://doi.org/10.1007/3-540-44750-4_2

[11] M. Bellare, P. Rogaway, and D. Wagner. 2003. EAX: A Conventional Authenticated-Encryption Mode. Cryptology ePrint Archive, Report 2003/069. https://eprint.iacr.org/2003/069.

[12] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Proceedings of the 9th International Workshop on Public Key Cryptography (PKC 2006)*. 207–228. https://doi.org/10.1007/11745853

[13] Daniel J. Bernstein. 2008. ChaCha, a variant of Salsa20. https://cr.yp.to/chacha/chacha-20080128.pdf. Last accessed: July 30, 2018.

[14] Daniel J. Bernstein. 2008. The Salsa20 Family of Stream Ciphers. In *New Stream Cipher Designs*, Matthew Robshaw and Olivier Billet (Eds.). Springer, 84–97. https://doi.org/10.1007/978-3-540-68351-3_8

[15] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of Cryptographic Engineering* 2, 2 (Sept. 2012), 77–89. https://doi.org/10.1007/s13389-012-0027-1

[16] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2011. NaCl: Networking and Cryptography Library. https://nacl.cr.yp.to/index.html. Last accessed: July 31, 2016.

[17] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *Proceedings of the 2nd International Conference on Cryptology and Information Security in Latin America (LATINCRYPT '12)*. Springer, 159–176. https://doi.org/10.1007/978-3-642-33481-8_9

[18] Bruno Blanchet. 2008. A Computationally Sound Mechanized Prover for Security Protocols. *IEEE Transactions on Dependable and Secure Computing* 5, 4 (October 2008), 193–207. https://doi.org/10.1109/TDSC.2007.1005

[19] Bruno Blanchet. 2011. Using Horn Clauses for Analyzing Security Protocols. In *Formal Models and Techniques for Analyzing Security Protocols*. Cryptology and Information Security Series, Vol. 5. IOS Press, 86–111.

[20] Bruno Blanchet. 2014. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In *Foundations of Security Analysis and Design VII, FOSAD Tutorial Lectures*. Lecture Notes in Computer Science, Vol. 8604. Springer, 54–87.

[21] Bruno Blanchet. 2018. ProVerif: Cryptographic Protocol Verifier in the Formal Model. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/. Latest release: 2.00. Last accessed: August 6, 2018.

[22] Bruno Blanchet and David Cadé. 2018. CryptoVerif: Cryptographic Protocol Verifier in the Computational Model. http://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/cryptoverif.html. Latest release: 2.00. Last accessed: August 6, 2018.

[23] Ran Canetti and Hugo Krawczyk. 2001. Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT '01)*. Springer, 453–474. http://dx.doi.org/10.1007/3-540-44987-6

[24] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. 2016. Evaluation of Cryptography Usage in Android Applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (Formerly BIONETICS) (BICT '15)*. 83–90. https://doi.org/10.4108/eai.3-12-2015.2262471

[25] Ronald Cramer and Victor Shoup. 2001. Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack. *SIAM J. Comput.* 33 (2001), 167–226.

[26] C.J.F. Cremers. 2008. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008) (Lecture Notes in Computer Science)*, Vol. 5123. Springer, 414–418. https://doi.org/10.1007/978-3-540-70545-1_38

[27] Cas Cremers. 2014. The Scyther Tool for the Symbolic Analysis of Security Protocols. https://github.com/cascremers/scyther. Latest release: 1.1.3. Last accessed: August 6, 2018.

[28] Cas Cremers and Sjouke Mauw. 2012. *Operational Semantics and Verification of Security Protocols*. Springer. https://doi.org/10.1007/978-3-540-78636-8

[29] Dorothy E. Denning and Giovanni Maria Sacco. 1981. Timestamps in Key Distribution Protocols. *Commun. ACM* 24, 8 (Aug. 1981), 533–536. https://doi.org/10.1145/358722.358740

[30] Arkajit Dey and Stephen Weis. 2015. Keyczar: A Cryptographic Toolkit. http://keyczar.googlecode.com/files/keyczar05b.pdf.

[31] T. Dierks and E. Rescorla. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Internet Engineering Task Force. https://tools.ietf.org/html/rfc5246.

[32] Morris J. Dworkin. 2001. *Recommendation for Block Cipher Modes of Operation: Method and Techniques*. Special Publication 800-38A. National Institute of Standards & Technology.

[33] Morris J. Dworkin. 2004. *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. Special Publication 800-38C. National Institute of Standards & Technology.

[34] Morris J. Dworkin. 2007. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Special Publication 800-38D. National Institute of Standards & Technology.

[35] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*. 73–84. https://doi.org/10.1145/2508859.2516693

[36] Helder Eijs. 2018. PyCryptodome: Python Package of Low-Level Cryptographic Primitives. https://pycryptodome.readthedocs.io/en/latest/. Latest release: 3.7.0. Last accessed: November 16, 2018.

[37] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. 50–61. https://doi.org/10.1145/2382196.2382205

[38] Frank Denis. 2017. The Sodium crypto library (libsodium). https://download.libsodium.org/doc/. Latest release: 1.0.16. Last accessed: August 6, 2018.

[39] D. Harkins. 2008. *Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)*. RFC 5297. Internet Engineering Task Force. https://tools.ietf.org/html/rfc5297.

[40] R. Housley. 2009. *Cryptographic Message Syntax (CMS)*. RFC 5652. Internet Engineering Task Force. https://tools.ietf.org/html/rfc5652.

[41] T. Kivinen and M. Kojo. 2003. *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*. RFC 3526. Internet Engineering Task Force. https://tools.ietf.org/html/rfc3526.

[42] H. Krawczyk and P. Eronen. 2010. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. Internet Engineering Task Force. https://tools.ietf.org/html/rfc5869.

[43] T. Krovetz and P. Rogaway. 2014. *The OCB Authenticated-Encryption Algorithm*. RFC 7253. Internet Engineering Task Force. https://tools.ietf.org/html/rfc7253.

[44] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 931–936. http://dl.acm.org/citation.cfm?id=3155562.3155681

[45] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why Does Cryptographic Software Fail? A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys '14)*. ACM, Article 7, 7 pages. https://doi.org/10.1145/2637166.2637237

[46] M. Lepinski and S. Kent. 2008. *Additional Diffie-Hellman Groups for Use with IETF Standards*. RFC 5114. Internet Engineering Task Force. https://tools.ietf.org/html/rfc5114.

[47] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2014. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In *Proceedings of the 8th International Conference on Network and System Security (NSS 2014)*. Springer, 349–362. https://doi.org/10.1007/978-3-319-11698-3_27

[48] Bo Lin and Yanhong A. Liu. 2018. DistAlgo: A Language for Distributed Algorithms. https://github.com/distalgo/distalgo. Beta release September 27, 2014. Latest release September 18, 2018.

[49] Dwayne Litzenberger. 2013. PyCrypto: The Python Cryptography Toolkit. https://www.dlitz.net/software/pycrypto/. Latest release: 2.6.1. Last accessed: August 6, 2018.

[50] Yanhong A Liu, Scott D Stoller, and Bo Lin. 2017. From Clarity to Efficiency for Distributed Algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 3 (2017), 12.

[51] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. 2012. From Clarity to Efficiency for Distributed Algorithms. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*. 395–410.

[52] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS '16)*. 711–722. https://doi.org/10.1145/2897845.2897896

[53] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The Tamarin Prover for the Symbolic Analysis of Security Protocols. In *Proceedings of Computer Aided Verification, 25th International Conference, CAV 2013 (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[54] Simon Meier, Benedikt Schmidt, Cas Cremers, Cedric Staub, Jannik Dreier, and Ralf Sasse. 2018. The Tamarin Prover For Security Protocol Verification. https://github.com/tamarin-prover/tamarin-prover. Latest release: 1.4.0. Last accessed: August 6, 2018.

[55] Microsoft. 2018. System.Security.Cryptography: .NET cryptography library for C#. https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography?view=netframework-4.7.1. Latest release: 4.7.2. Last accessed: November 13, 2018.

[56] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. 2016. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Internet Engineering Task Force. https://tools.ietf.org/html/rfc8017.

[57] Moxie Marlinspike. 2013. Advanced Cryptographic Ratcheting. https://signal.org/blog/advanced-ratcheting/. Last accessed: May 9, 2018.

[58] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH Key Agreement Protocol. https://signal.org/docs/specifications/x3dh/. Last accessed: May 9, 2018.

[59] Roger M. Needham and Michael D. Schroeder. 1978. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM* 21, 12 (Dec. 1978), 993–999. https://doi.org/10.1145/359657.359659

[60] R M Needham and M D Schroeder. 1987. Authentication Revisited. *SIGOPS Oper. Syst. Rev.* 21, 1 (Jan. 1987), 7–7. https://doi.org/10.1145/24592.24593

[61] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. 2005. *The Kerberos Network Authentication Service (V5)*. RFC 4120. Internet Engineering Task Force. http://www.rfc-editor.org/rfc/rfc4120.txt.

[62] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 1065–1077. https://doi.org/10.1145/3133956.3133977

[63] NIST. 2001. *Advanced Encryption Standard (AES)*. FIPS Publication 197. National Institute of Standards and Technology. https://csrc.nist.gov/publications/detail/fips/197/final

[64] NIST. 2013. *Digital Signature Standard (DSS)*. FIPS Publication 186-4. National Institute of Standards and Technology. https://csrc.nist.gov/publications/detail/fips/186/4/final

[65] NIST. 2015. *Secure Hash Standard (SHS)*. FIPS Publication 180-4. National Institute of Standards and Technology. https://csrc.nist.gov/publications/detail/fips/180/4/final

[66] NIST. 2016. *Recommendation for Key Management, Part 1: General*. Special Publication 800-57 Part 1 Revision 4. National Institute of Standards and Technology. https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final

[67] NIST. 2017. *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*. Special Publication 800-67 Revision 2. National Institute of Standards and Technology. https://csrc.nist.gov/publications/detail/sp/800-67/rev-2/final

[68] Open Whisper Systems. 2016. libsignal-protocol-java. https://github.com/signalapp/libsignal-protocol-java. Latest release: 2.3.0. Last accessed: May 9, 2018.

[69] Oracle 2018. *Java Cryptography Architecture (JCA) Reference Guide*. Oracle. https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

[70] William R. Price. 1992. Issues to Consider When Using Evaluated Products to Implement Secure Mission Systems. In *Proceedings of the 15th National Computer Security Conference*, Vol. 1. NIST, 292–299. https://csrc.nist.gov/publications/detail/conference-paper/1992/10/13/proceedings-15th-national-computer-security-conference-1992

[71] Python Cryptographic Authority. 2018. pyca/cryptography. https://github.com/pyca/cryptography. Latest release: 2.14. Last accessed: July 30, 2018.

[72] Python Cryptographic Authority. 2018. PyNaCl. https://github.com/pyca/pynacl. Latest release: 1.3.0. Last accessed: November 16, 2018.

[73] Bruce Schneier. 1994. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In *Proceedings of the International Workshop on Fast Software Encryption (FSE 1993)*. Springer, 191–204. https://doi.org/10.1007/3-540-58108-1_24

[74] Victor Shoup. 1999. On Formal Models for Secure Key Exchange. http://www.shoup.net/papers/skey.pdf. Technical Report.

[75] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. 2014. Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *Proceedings of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)*. 75–80. https://doi.org/10.1109/DASC.2014.22

[76] Ng Pheng Siong and Matej Cepl. 2018. M2Crypto. https://gitlab.com/m2crypto/m2crypto. Latest release: 0.31.0 Last accessed: November 16, 2018.

[77] SPORE. 2003. Security Protocols Open Repository (SPORE). http://www.lsv.fr/Software/spore. Last accessed: May 19, 2017.

[78] Google Security Team. 2016. Keyczar: Open Source Cryptographic Toolkit. https://github.com/google/keyczar. Latest release: 0.71j (Java), 0.716 (Python), 0.71a (C++). Last accessed: July 30, 2018.

[79] Trevor Perrin and Moxie Marlinspike. 2016. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/. Last accessed: May 9, 2018.