Comparing the Usability of Cryptographic APIs

Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel*,
Doowon Kim†, Michelle L. Mazurek†, and Christian Stransky
CISPA, Saarland University; *National Institute of Standards and Technology; †University of Maryland, College Park

***Abstract*—Potentially dangerous cryptography errors are well-documented in many applications. Conventional wisdom suggests that many of these errors are caused by cryptographic Application Programming Interfaces (APIs) that are too complicated, have insecure defaults, or are poorly documented. To address this problem, researchers have created several cryptographic libraries that they claim are more usable; however, none of these libraries have been empirically evaluated for their ability to promote more secure development. This paper is the first to examine both how and why the design and resulting usability of different cryptographic libraries affects the security of code written with them, with the goal of understanding how to build effective future libraries. We conducted a controlled experiment in which 256 Python developers recruited from GitHub attempt common tasks involving symmetric and asymmetric cryptography using one of five different APIs. We examine their resulting code for functional correctness and security, and compare their results to their self-reported sentiment about their assigned library. Our results suggest that while APIs designed for simplicity can provide security benefits—reducing the decision space, as expected, prevents choice of insecure parameters—simplicity is not enough. Poor documentation, missing code examples, and a lack of auxiliary features such as secure key storage, caused even participants assigned to simplified libraries to struggle with both basic functional correctness and security. Surprisingly, the availability of comprehensive documentation and easy-to-use code examples seems to compensate for more complicated APIs in terms of functionally correct results and participant reactions; however, this did not extend to security results. We find it particularly concerning that for about 20% of functionally correct tasks, across libraries, participants believed their code was secure when it was not.**

Our results suggest that while new cryptographic libraries that want to promote effective security should offer a simple, convenient interface, this is not enough: they should also, and perhaps more importantly, ensure support for a broad range of common tasks and provide accessible documentation with secure, easy-to-use code examples.

I. Introduction

Today's connected digital economy and culture run on a foundation of cryptography, which both authenticates remote parties to each other and secures private communications. Cryptographic errors can jeopardize people's finances, publicize their private information, and even put political activists at risk [1]. Despite this critical importance, cryptographic errors have been well documented for decades, in both production applications and widely used developer libraries [2]–[5].

Many researchers have used static and dynamic analysis techniques to identify and investigate cryptographic errors in source code or binaries [2]–[6]. This approach is extremely valuable for illustrating the pervasiveness of cryptographic errors, and for identifying the kinds of errors seen most frequently in practice, but it cannot reveal root causes. Conventional wisdom in the security community suggests these errors proliferate in large part because cryptography is so difficult for non-experts to get right. In particular, libraries and Application Programming Interfaces (APIs) are widely seen as being complex, with many confusing options and poorly chosen defaults (e.g. [7]). Recently, cryptographers have created new libraries with the goal of addressing developer usability by simplifying the API and establishing secure defaults [8], [9]. To our knowledge, however, none of these libraries have been empirically evaluated for usability. To this end, we conduct a controlled experiment with real developers to investigate root causes and compare different cryptographic APIs. While it may seem obvious that simpler is better, a more in-depth evaluation can be used to reveal where these libraries succeed at their objectives and where they fall short. Further, by understanding root causes of success and failure, we can develop a blueprint for future libraries.

This paper presents the first empirical comparison of several cryptographic libraries. Using Python as common implementation language, we conducted a 256-person, between-subjects online study comparing five Python cryptographic libraries chosen to represent a range of popularity and usability: cryptography.io, Keyczar, PyNaCl, M2Crypto and PyCrypto. Open-source Python developers completed a short set of cryptographic programming tasks, using either symmetric or asymmetric primitives, and using one of the five libraries. We evaluate participants' code for functional correctness and security, and also collect their self-reported sentiment toward the usability of the library. Taken together, the resulting data allows us to compare the libraries for usability, broadly defined to include ability to create working code, effective security in practice (when used by primarily non-security-expert developers), and participant satisfaction. By using a controlled, random-assignment experiment, we can compare the libraries directly and identify root causes of errors, without confounds related to the many reasons particular developers may choose particular libraries for their real projects.

We find that simplicity of individual mechanisms in an API does not assure that the API is, in fact, usable. Instead, the stronger predictors of participants producing working code

The identification of a commercial product or trade name does not imply endorsement or recommendation by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

were the quality of documentation, and in particular whether examples of working code were available on the Internet, within or outside the provided documentation. Surprisingly, we also found that the participant's Python experience level, security background, and experience with their assigned library did not significantly predict the functionality of the code that they created. None of the libraries were rated as objectively highly usable, but PyCrypto, a complex API with relatively strong documentation, was rated significantly more usable than Keyczar, a simple API with poor documentation.

On the other hand, with some important caveats, simplified APIs did seem to promote better security results. As might be expected, reducing the number of choices developers must make (for example, key size or encryption mode of operation) also reduces their opportunity to choose incorrect parameters. Python experience level was not significantly correlated with security results, but participants with a security background were more likely to produce code that was, in fact, secure. Nevertheless, the overall security results were somewhat disappointing. A notable source of problems was APIs that did not easily support important auxiliary tasks, such as secure key storage. Perhaps of most concern, 20% of functional solutions were rated secure by the participant who developed them but insecure according to our evaluation; this suggests an important failure to communicate important security ideas or warn about insecure decisions.

II. Related Work

We discuss related work in four key areas: measuring cryptography problems in deployed code; investigating how developers interact with cryptographic APIs; attempts at developing more usable cryptographic libraries and related tools; and approaches to evaluating API usability more generally.

Cryptography problems in real code. Researchers have identified misuses of cryptography in deployed code. Egele et al. examined more than 11,000 deployed Android apps that use cryptography and found that nearly 90% contained at least one of six common cryptography errors [5]. Fahl et al. and Onwuzurike et al. also analyzed Android apps, and found that a large number did not correctly implement the Trusted Layer Security (TLS) protocol, potentially leading to security vulnerabilities to Man-In-The-Middle (MITM) attacks [10]–[15]. Likewise, a study examining Apple's iOS apps revealed that many were vulnerable to MITM attacks because of incorrect certificate validation during TLS connection establishment [16]. Other researchers specifically examined mobile banking applications and found a plethora of potentially exploitable cryptographic errors [4]. Lazar et al. examined cryptography-related vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database and found more than 80% resulted from errors at the application level [17]. In all of these cases, weak ciphers and insufficient randomness were common problems; in this paper, we test the hypothesis that these problems are strongly affected by API design. Georgiev et al. identified many certificate-validation errors in applications and libraries; the authors attribute many of these vulnerabilities to poorly designed APIs and libraries with too many confusing options [3].

Interacting with cryptographic APIs. Others have investigated how developers interact with cryptographic APIs. Nadi et al. manually examined the top 100 Java cryptography posts on Stack Overflow and found that a majority of problems were related to API complexity rather than a lack of domain knowledge [18]. Follow-up surveys of some Stack Overflow users who had asked questions and of Java developers more generally confirmed that API complexity and poor documentation are common barriers in cryptographic API use. In this paper, we compare different APIs to measure their relative difficulty of use. Relatedly, Acar et al. examined how use of different documentation resources affects developers' security decisions, including decisions about certificate validation [19]; we compare different APIs rather than different sources of help.

Making cryptography more usable. Several cryptographic APIs have been designed with usability in mind. The designers of NaCl (Networking and Cryptographic library, pronounced "salt") describe how their design decisions are intended to promote usability, in large part by reducing the number of decisions a developer must make, but do not empirically evaluate its usability [9]. In this work, we empirically compare NaCl to more traditional APIs, as well as to non-academic libraries that also claim usability benefits (e.g., cryptography.io [8]).

Rather than a new API, Arzt et al. present an Eclipse plugin that produces correct code templates based on high-level requirements identified by the developer [20]. This approach can make working with existing APIs easier; however, it is orthogonal to the question of how APIs do or do not encourage secure practices. Indela et al. suggest using design patterns to describe high-level *semantic APIs* for goals that require cryptography, such as establishing a secure connection or storing data securely [21]. This approach is complementary to improving cryptographic libraries that underlie such patterns.

Evaluating APIs, security and otherwise. Many software engineering researchers have examined what makes an API usable. Myers and Stylos provide a broad overview of how to evaluate API usability, with reference to Nielsen's general usability guidelines as well as the Cognitive Dimensions framework [22]–[24]. Henning and Bloch separately provide sets of maxims for improving API design [25], [26]. Smith and Green proposed similar high-level guidelines specific to security APIs [27]. We adapt guidelines from these various sources to evaluate the APIs we examine.

Concurrent with our work, Gorski and Iacono [28] use an extensive literature review to formulate high-level technical and usability criteria along which security-relevant APIs should be designed, calling for further work on evaluating adherence to these principles. Also concurrent to our work, Wijayarathna et al. develop a set of questions about security APIs based on the above guidelines, resulting in a questionnaire similar to the one we developed and used in this work [29].

Oliveira et al. conducted a lab study to examine the security mindset of developers. They found that security is not a priority in the standard developer's mindset, but that detailed priming for security issues helps [30]. Wurster and Van Oorschot recommend assuming that developers will not prioritize security unless incentivized or forced to, and suggest mandating security tools, rewarding secure coding practices, and ensuring that secure tools and APIs are more usable and attractive than less secure ones [31]. Our work focuses on how choice of library affects developers who have already decided to interact with a cryptographic API and have been primed for the importance of security to their task.

Finifter, Wagner and Prechelt compared the security of two web applications built to the same specification but with different frameworks. They found that automatic framework-level support for mitigating certain vulnerabilities improved overall security, while manual framework supports were readily forgotten or neglected [32], [33].

Researchers have also conducted empirical studies of API usability in different domains, including comparing APIs for configuration [34], considering how assigning methods to classes affects usability [35], and analyzing the usability of the factory pattern [36]. Piccioni et al. examined the usability of a persistence library using a method similar to the one we use in this work, with exit interview questions structured around the Cognitive Dimensions framework [37]. They successfully identify usability failures of the examined API, and their results emphasize the critical importance of accurate, unambiguous and self-contained documentation to API usability. Burns et al. provide a preliminary survey of work evaluating APIs empirically [38].

III. Study Design

We designed an online, between-subjects study to compare how effectively developers could quickly write correct, secure code using different cryptographic libraries. We recruited developers with demonstrated Python experience (on GitHub) for an online study.

Participants were assigned to complete a short set of programming tasks using either symmetric- or asymmetric-key cryptography, using one of five Python cryptographic libraries. Assignment to one of the resulting 10 conditions was initially random, with counterbalancing to ensure roughly equivalent participant counts starting each condition. As the study progressed, however, it became clear that dropout rates varied widely by condition (see Section IV-C for details), so we weighted the random assignment to favor conditions with higher dropout rates.

Within each condition, task order was randomized. Symmetric participants were either given a key generation, then an encryption/decryption task, or vice-versa. Asymmetric participants were assigned a key generation task, an encryption/decryption task, and a certificate validation task, according to a latin square ordering.

After finishing the tasks, participants completed a brief exit survey about the experience. We examined participants' submitted code for functional correctness and security. The study was approved by our institutions' ethics review boards.

A. Language selection

We chose to use Python as the programming language for our experiment because it is widely used across many communities and has support for all kinds of security-related APIs, including cryptography. As a bonus, Python is easy to read and write and is widely used among both beginners and experienced programmers. Indeed, Python is the third most popular language on GitHub, trailing JavaScript and Java [39]. Therefore, we reasoned that there would be many Python developers to recruit for our study.

B. Cryptographic library identification

Next, we performed a series of Internet searches to identify possible cryptographic libraries that we could use in our study. We were agnostic to library implementation language, performance, and third-party certification: all that mattered was that the library could be called from Python language bindings. At this point, we decided to use the Python 2.7 programming language because several Python cryptographic libraries did not support Python 3.

We selected five Python libraries to empirically compare based on a combination of their popularity, their suitability for the range of tasks we were interested in, and our desire to compare libraries that were and were not designed with usability in mind. Table I lists details of these features for the libraries we examined.

We selected three libraries whose documentation claims they were designed for usability and that each handle (most of) the tasks we were interested in: cryptography.io, Keyczar, and PyNaCl. cryptography.io describes itself as "cryptography for humans" [8], Keyczar is "designed to make it easier and safer for developers to use cryptography" [40], and PyNaCl is a Python binding for NaCl, a crypto library designed to avoid "disaster" in part via simplified APIs [9]. pysodium is a potential alternative to PyNaCl; although pysodium is very slightly more popular, it is still beta and has no included documentation, so we selected PyNaCl.

For comparison, we also selected two libraries that do not make usability claims: PyCrypto and M2Crypto. PyCrypto is the most popular general-purpose Python crypto library we found, and the closest thing to a "default" Python crypto library that exists. M2Crypto is a Python binding for the venerable OpenSSL library, which is frequently criticized for its lack of usability. pyOpenSSL is both more popular than M2Crypto and the official OpenSSL [41] binding for Python; however, it lacks support for symmetric and asymmetric encryption, which was a major part of our study, so we opted for M2Crypto instead. We provide further details about the features and documentation of the libraries we selected in Section III-F.

We excluded libraries that include few of the features we were interested in, or that have negligible popularity. We excluded PyCryptodome as a less popular replacement for

		Sym		Asym						
		Key generation	Encryption	Key generation	Encryption	KDF	Digital sig.	X.509	Usability claims	Downloads
PyCrypto	[42]	●	●	●	●	●	●	●	○	25 149 446
cryptography.io	[8]	●	●	●	●	●	●	●	●	10 481 277
M2Crypto	[43]	●	●	●	●	●	●	●	○	2 369 827
Keyczar	[44]	●	●	●	●	○	●	○	●	595 277
PyNaCl	[45]	●	●	●	●	○	●	○	●	46 013
pyOpenSSL	[46]	○	○	○	○	○	●	●	○	10 188 101
tlslite	[47]	○	○	○	○	○	○	●	○	641 488
bcrypt	[48]	○	○	○	○	○	○	○	○	536 851
gnupg	[49]	○	●	●	●	○	●	○	○	189 851
pycryptopp	[50]	●	●	●	○	○	●	○	○	140 703
scrypt	[51]	○	●	○	○	○	○	○	○	140 446
simple-crypt	[52]	●	●	○	○	●	○	○	●	112 254
pysodium	[53]	●	●	●	●	○	●	○	●	49 275
ed25519	[54]	○	○	●	○	○	●	○	○	29 670
pyaes	[55]	○	●	○	○	○	○	○	○	19 091
PyCryptodome	[56]	●	●	●	●	●	●	○	○	16 960
PyMe	[57]	○	○	●	●	○	●	○	○	2 489
pyDes	[58]	○	●	○	○	○	○	○	○	? †
tls	[59]	○	○	○	○	○	○	○	●	? †

● = applies; ○ = does not apply

TABLE I
Cryptography-related Python libraries and their features, ordered by popularity. The top section includes the libraries we tested. Download counts as of May 2016 were taken from the PyPI ranking website (http://pypi-ranking.info). †No download statistics available.

PyCrypto, gnupg for its limited support for encryption (mainly in the context of email), pycryptopp as it was deprecated as of January 2016, and simple-crypt as it does not support asymmetric cryptography.

In tables and figures throughout the paper, we order the libraries as follows: PyCrypto first as the most popular, then M2Crypto as the other library without usability claims, then the three libraries with usability claims.

C. Recruitment and framing

To maintain ecological validity, we wanted to recruit developers who actively use Python. To find such developers, we conducted a systematic analysis of Python contributors on the popular GitHub collaborative source code management service.

We extracted all Python projects from the GitHub Archive database [60] between GitHub's launch in April 2008 and February 2016, giving us 749 609 projects in total. We randomly sampled 100 000 of these repositories and cloned them. Using this random sample, we extracted email addresses of 50 000 randomly chosen Python committers. These committers served as a source pool for our recruiting.

We emailed these developers in batches, asking them to participate in a study exploring how developers use Python libraries. We did not mention cryptography or security in the recruitment message. We mentioned that we would not be able to compensate them, but the email offered a link to learn more about the study and a link to remove the email address from any further communication about our research. Each contacted developer was assigned a unique pseudonymous identifier (ID) to allow us to correlate their study participation to their GitHub statistics separately from their email address.

Recipients who clicked the link to participate in the study were directed to a landing page containing a consent form. After affirming they were over 18, consented to the study, and were comfortable with participating in the study in English, they were introduced to the study framing. We asked participants to imagine they were developing code for an app called CitizenMeasure, "a new global monitoring system that will allow citizen-scientists to travel to remote locations and make measurements about such issues as water pollution, deforestation, child labor, and human trafficking. Please keep in mind that our citizen-scientists may be operating in locations that are potentially dangerous, collecting information that powerful interests want kept secret. Our citizen scientists may have their devices confiscated and hacked." We hoped that this framing would pique participants' interest and motivate them to make a strong effort to write secure code. We also provided brief instructions for using the study infrastructure, which we describe next.

D. Experimental infrastructure

After reading the study introduction and framing, participants were redirected to the tasks themselves. Our aim was to conduct an online developer study in which real developers would write and test real cryptographic code in our environment. We wanted to capture the code that they typed and their program runs. We wanted to control the study environment (Python version, available libraries) and collect data about their progress in real time. To achieve this, we used Jupyter Notebook [61], which allowed participants to write and run Python code in their browser, using the Python installation from our server. We instrumented the notebook to frequently snapshot the participant's code, as well as to detect and store copy&paste events. All this information was stored on the server.

We configured Notebook (version 4.2.1) with Python 2.7.11 and all five tested cryptographic libraries. To prevent interference between participants, each participant was assigned to a Notebook running on a separate Amazon Web Service (AWS) instance. We maintained a pool of prepared instances so that each new participant could begin without waiting for an instance to boot. Instances were shut down when each participant finished, to avoid between-subjects contamination.

Tasks were shown one at a time, with a progress indicator showing that the participant had completed, e.g., 1 of 3 tasks. For each task, participants were given buttons to "Run and test" their code, and to move on using "Solved, next task" or “Not solved, but next task.” After each button press, we stored the participant’s current code, along with metadata like timing, in a remote database. An example Notebook is shown in Figure 1.

Allowing participants to write and execute Python code presents serious security concerns. To mitigate this, we removed all unnecessary packages from the AWS image. We used the AWS firewall to restrict incoming traffic to port 80 and prevent outgoing traffic other than to our study database, which was password protected and restricted to sanitized insert commands. All instances were shut down within 4 hours of the last observed participant activity.

E. Task design

We designed tasks that were short enough so that the uncompensated participants would be likely to complete them before losing interest, but still complex enough to be interesting and allow for some mistakes. Most importantly, we designed tasks to model real world problems that Python developers could reasonably be expected to encounter in their professional career. We chose two symmetric-encryption tasks: generating an encryption key and storing it securely in a password-protected file, and using the key to encrypt and decrypt text. We chose three asymmetric tasks: generating a key pair and storing the private key securely, using the public key to encrypt and the private key to decrypt, and validating an X.509 certificate.

Most of the libraries we chose support most of these tasks (Table II). Unfortunately, task coverage by the libraries was not uniform: Keyczar and PyNaCl do not support secure key storage. The Keyczar documentation encourages generating keys at the command line; this can be worked around in the API, but it is not straightforward to do so. Keyczar and PyNaCl do not support certificate validation directly, but it is possible to extract the public key and manually verify the signature. Finally, PyCrypto does not support certificate validation at all.

To account for cases where the library does not fully support the task, we offered participants the option to skip a task.

For each task, participants were provided with stub code and some commented instructions. These stubs were designed to make the task clear and ensure the results could be easily evaluated, without providing too much scaffolding. We also provided a main method pre-filled with code to test the provided stubs. This helped orient participants and saved time, but it did prevent us from learning how participants might have designed their own tests.

We also asked participants to please use only the included documentation for their assigned library, if at all possible, and to report (in comments) any additional documentation resources they consulted.

F. Python cryptographic libraries we included

We briefly review the available features and documentation for each library we selected for our experiment (Table II).

Library	Current Version	Designed for Usability	Symmetric Key Generation	Symmetric Encryption/Decryption	Secure Symmetric Key Storage	Asymmetric Key Generation	Asymmetric Encryption/Decryption	Secure Asymmetric Key Storage	Certificate Validation
PyCrypto	2.6.1	○	●	●	●	●	●	●	○
M2Crypto	0.25.1	○	●	●	●	●	●	●	●
cryptography.io	1.4	●	●	●	●	●	●	●	●
Keyczar	0.716	●	◐	●	○	◐	●	○	◐
PyNaCl	1.0.1	●	●	●	○	●	●	○	◐

● = fully applies; ◐ = partly applies; ○ = does not apply

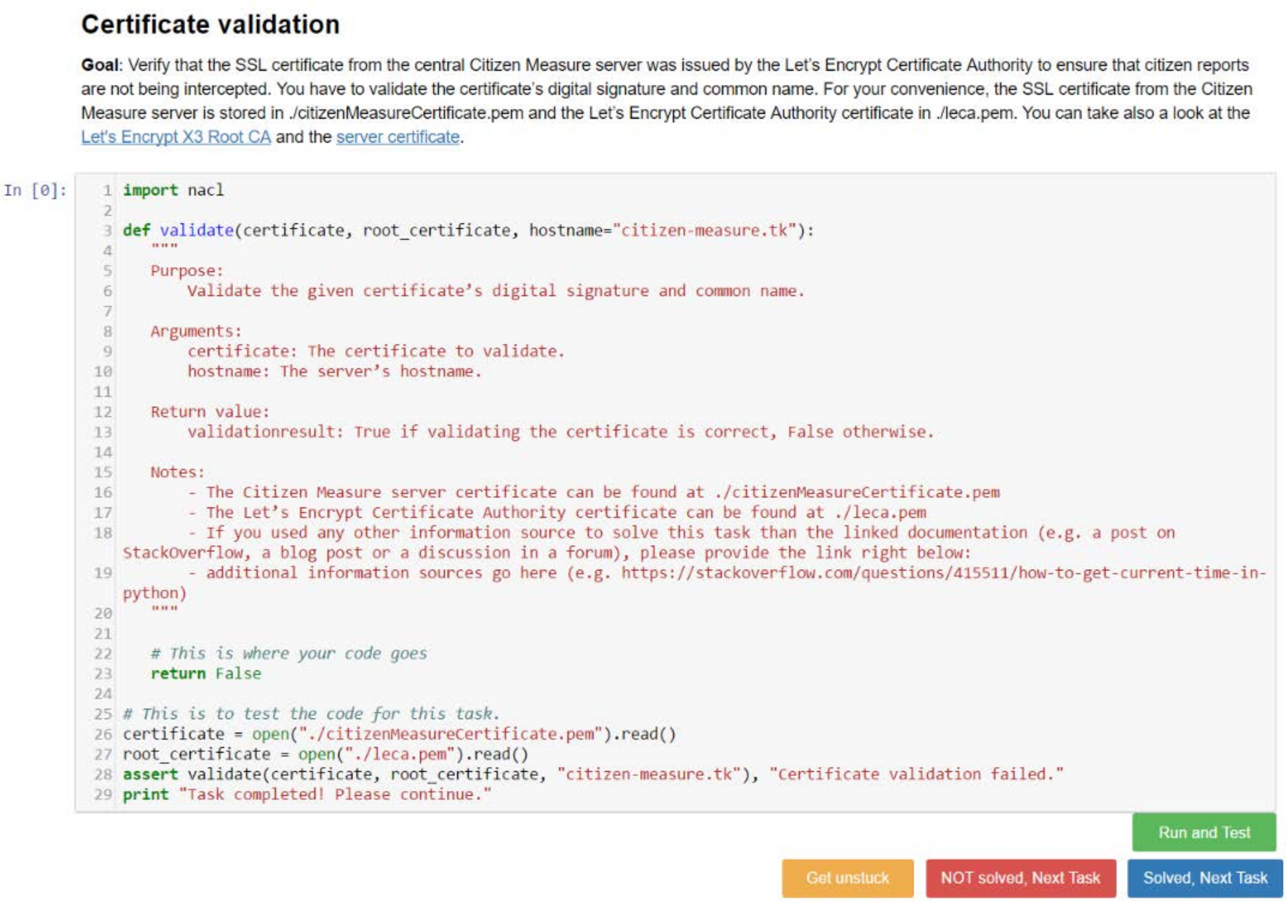
TABLE II
Features and popularity for the five cryptography libraries we tested. Popularity data was updated as of Aug. 11, 2016.

PyCrypto. The Python cryptographic toolkit PyCrypto [42] is Python’s most popular cryptographic library. Developers can choose among several encryption and hashing algorithms and modes of operation, and may provide initialization vectors (IVs).

PyCrypto comes with primarily auto-generated documentation that includes minimal code examples. The documentation recommends the Advanced Encryption Standard (AES) and provides an example, but also describes the weaker Data Encryption Standard (DES) as cryptographically secure. The documentation warns against weak exclusive-or (XOR) encryption. However, the documentation does not warn against using the default Electronic Code Book (ECB) mode, or the default empty IV, neither of which is secure.

M2Crypto. M2Crypto [43] is a binding to the well-known OpenSSL library that is more complete than alternative bindings such as pyOpenSSL. Although development on M2Crypto has largely ceased, the library is still widely used, and there is ample documentation and online usage examples, so we included it. M2Crypto supports all of the tasks we tested, including X.509 certificate handling. Developers are required to choose algorithms, modes of operation, and initialization vectors. M2Crypto comes with automatically generated documentation that includes no code examples or comments on the security of cryptographic algorithms and modes.

cryptography.io. cryptography.io has a stated goal of providing more usable security than other libraries by emphasizing secure algorithms, high-level methods, safe defaults, and good documentation [8]. It supports symmetric and asymmetric encryption as well as X.509 certificate handling. The documentation includes code examples that include secure options, with context for how they should be used. cryptography.io provides a high-level interface for some cryptographic tasks (such



Certificate validation

Goal: Verify that the SSL certificate from the central Citizen Measure server was issued by the Let's Encrypt Certificate Authority to ensure that citizen reports are not being intercepted. You have to validate the certificate's digital signature and common name. For your convenience, the SSL certificate from the Citizen Measure server is stored in ./citizenMeasureCertificate.pem and the Let's Encrypt Certificate Authority certificate in ./leca.pem. You can take also a look at the Let's Encrypt X3 Root CA and the server certificate.

```
In [0]:
import nacl

def validate(certificate, root_certificate, hostname="citizen-measure.tk"):
    """
    Purpose:
        Validate the given certificate's digital signature and common name.

    Arguments:
        certificate: The certificate to validate.
        hostname: The server's hostname.

    Return value:
        validationresult: True if validating the certificate is correct, False otherwise.

    Notes:
        - The Citizen Measure server certificate can be found at ./citizenMeasureCertificate.pem
        - The Let's Encrypt Certificate Authority certificate can be found at ./leca.pem
        - If you used any other information source to solve this task than the linked documentation (e.g. a post on
StackOverflow, a blog post or a discussion in a forum), please provide the link right below:
        - additional information sources go here (e.g. https://stackoverflow.com/questions/415511/how-to-get-current-time-in-
python)
    """

    # This is where your code goes
    return False

# This is to test the code for this task.
certificate = open("./citizenMeasureCertificate.pem").read()
root_certificate = open("./leca.pem").read()
assert validate(certificate, root_certificate, "citizen-measure.tk"), "Certificate validation failed."
print "Task completed! Please continue."
```

Fig. 1. An example of the study's task interface.

as symmetric key generation and encryption); this interface does not require developers to choose any security-sensitive parameters. The library also includes a lower-level interface, necessary for some asymmetric tasks and for encrypted key storage; this low-level interface does require developers to specify parameters such as algorithm and salt.

Keyczar. The library aims to make it easier to safely use cryptography, so that developers do not accidentally expose key material, use weak key lengths or deprecated algorithms, or improperly use cryptographic modes [40]. The documentation consists of an 11-page technical report that includes a few paragraphs regarding the program's design and a few abbreviated examples. Keyczar does not easily support X.509 certificate handling, encrypted key files, or password-based key derivation, but it does support digital signatures. There is no public API for key generation, but developers can generate keys by using an internal interface or by calling a provided command-line tool programmatically. Developers do not have to specify cryptographic algorithms, key sizes, or modes of operation.

PyNaCl. PyNaCl is a Python interface to libsodium [62], a cryptographic library designed with a focus on usability. The detailed documentation includes code examples with context for how to use them. PyNaCl supports both secure symmetric and asymmetric APIs without requiring the developer to choose cryptographic details, although the developer must provide a nonce. PyNaCl neither supports encrypted key storage nor password-based key derivation. X.509 certificate handling is also not supported directly; however, verifying digital signatures is supported.

G. *Exit survey*

Once all tasks had been completed or abandoned, the participant was directed to a short exit survey. We asked for their opinions about the tasks they had completed and the library they used, including the standard System Usability Scale (SUS) [63] score for the library. We also collected their demographics and programming experience. The participant's code for each task was displayed (imported from our database) for their reference with each question about that task.

We were specifically interested in the participants' opinions about the usability of the API. To this end, we collected the SUS score, but we wanted to also investigate in more depth. Prior work on API usability has suggested several concrete factors that affect an API's usability. We combined the cognitive dimensions framework [24] with usability suggestions from Nielsen and from Smith and Green [23], [27], and pulled out the factors that could most easily be evaluated via self-reporting from developers using the API. We transformed these factors into an 11-question scale (given in Appendix A) that focuses on the learnability of the API, the helpfulness of its documentation, the clarity of observed error messages, and other features. Our scale can be used to produce an overall score, as well as to target specific characteristics that impede the usability of each API. For this work, we treat this scale as exploratory; we correlate it with SUS and investigate its internal reliability in Section IV-F.

H. Evaluating participant solutions

We used the code submitted by our participants for each task, henceforth called a *solution*, as the basis for our analysis.

We evaluated each participant's solution to each task for both functional correctness and security. Every task was independently reviewed by two coders, using a codebook prepared ahead of time based on the capabilities of the libraries we evaluated. Differences between the two coders were adjudicated by a third coder, who updated the codebook accordingly. We briefly describe the codebook below.

Functionality. For each programming task, we assigned a participant a functionality score of 1 if the code ran without errors, passed the tests and completed the assigned task, or 0 if not.

Security. We assigned security scores only to those solutions which were graded as functional. To determine a security score, we considered several different security parameters. A participant's solution was marked secure (1) only if their solution was acceptable for every parameter; an error in any parameter resulted in a security score of 0.

Not all security parameters applied to all libraries, as some libraries do not allow users to make certain potentially insecure choices. Details of how the different security parameters applied to each library can be found in Table III. Whenever a given library requires a developer to make a secure choice for a given parameter, we assign a full circle; if that parameter is not applicable in that library, we assign an empty circle. For example, for symmetric encryption, PyCrypto participants had to specify an encryption algorithm, mode of operation and an initialization vector (three full circles). However, PyNaCl participants did not have to care about these cryptographic details (three empty circles).

For key generation, we checked key size and proper source of randomness for the key material. We selected an appropriate key size for a particular algorithm (e.g., for RSA we required at least 2 048-bit keys [64]). For key storage we checked if encryption keys were actually encrypted and if a proper encryption key was derived from the password we provided. Depending on the library and task type, encrypting cryptographic key material requires the application of a key derivation function such as PBDKF2 [65]. For libraries in which developers had to pick parameters for PBKDF2 manually (cf. Table III), we scored use of a static or empty salt, HMAC-SHA1 or below as the pseudorandom function, and less than 10 000 iterations as insecure [66]. For some libraries, participants had to select encryption parameters for one or more tasks; in these cases, we also scored the security of the chosen encryption algorithm, mode of operation, and initialization vector. For symmetric encryption, we scored ARC2, ARC4, Blowfish, (3)DES, and XOR as insecure, and AES as secure. We scored the ECB as an insecure mode of operation and scored Cipher Block Chaining (CBC), Counter Mode (CTR) and Cipher Feedback (CFB) as secure. Static, zero or empty initialization vectors were scored insecure. For asymmetric encryption we scored the use of OAEP/PKCS1 for padding as secure.

I. Limitations

As with any user study, our results should be interpreted in context. We chose an online study because it is difficult to recruit "real" developers (rather than students) for an in-person lab study at a reasonable cost. Choosing to conduct an online study allowed us less control over the study environment; however, it allowed us to recruit a geographically diverse sample. Because we targeted developers, we could not easily take advantage of services like Amazon's Mechanical Turk or survey sampling firms. Managing online study payments outside such infrastructures is very challenging; as a result, we did not offer compensation and instead asked participants to generously donate their time. As might be expected, the combination of unsolicited recruitment emails and no compensation led to a strong self-selection effect, and we expect that our results represent developers who are interested and motivated enough to participate. Comparing the full invited sample to the valid participants (see Section IV-A) suggests that indeed, more active GitHub users were more likely to participate. That said, these limitations apply across conditions, suggesting that comparisons between conditions are valid. Further, we found almost no results (Section IV-G) correlated with self-reported Python experience.

In any online study, some participants may not provide full effort, or may answer haphazardly. In this case, the lack of compensation reduces the motivation to answer in a manner that is not constructive; those who are not motivated will typically not participate. We attempt to remove any obviously low-quality data (e.g., responses that are entirely invective) before analysis, but cannot discriminate perfectly. Again, this limitation should apply across conditions without affecting condition comparisons.

Our study examines how developers use different cryptographic libraries. Developers who reach this point already recognize that they need encryption and have chosen to use an existing library rather than trying to develop their own mechanism; these are important obstacles to secure code that cannot be addressed by better library design. Nonetheless, we believe that evaluating and improving cryptographic libraries is a valuable step toward more secure development.

Finally, we are comparing libraries overall: this includes their API design and implementation as well as their documentation. The quality of both varies significantly across the libraries. Our results provide insight into the contributions

Symmetric

	Key Generation	Key Storage				Key Derivation			Encryption		
	Size	Plain/ Encrypted	Algorithm	Mode	IV	Salt	PRF	Iterations	Algorithm	Mode	IV
PyCrypto	●	●	●	●	●	●	●	●	●	●	●
M2Crypto	●	●	●	●	●	●	●	●	●	●	●
cryptography.io	●	●	○	○	○	●	●	●	○	○	○
Keyczar	○	●	○	○	○	●*	●*	●*	○	○	○
PyNaCl	○	●	○	○	○	●*	●*	●*	○	○	○

Asymmetric

	Key Generation		Key Storage				Encryption		Certificate Validation			
	Type	Size	Plain/ Encrypted	Algorithm	Mode	IV	Padding	Nonce	Signature Verification	Hostname Check	CA Check	Date Check
PyCrypto	●	●	●	○	○	○	●	○	●	●	●	●
M2Crypto	●	●	●	○	○	○	●	○	●	●	●	●
cryptography.io	●	●	●	○	○	○	●	○	●	●	●	●
Keyczar	○	○	●	●*	●*	●*	○	○	●	●	●	●
PyNaCl	○	○	●	●*	●*	●*	○	●	●	●	●	●

TABLE III
Security choices required by various libraries, as defined in our codebook. ● indicates the developer is required to make a secure choice, ○ indicates no such choice is required. Libraries that do not include a key derivation function, requiring the developer to fall back to Python's hashlib API, are indicated with *.

made by documentation and by API design to a library's overall success or failure, but future work is needed to further explore how the two operate independently.

IV. Study results

Study participants experienced very different rates of task completion, functional success, and security success as a function of which library they were assigned and whether they were assigned the symmetric or asymmetric tasks. Overall, we find that completion rate, functional success, and self-reported usability satisfaction showed similar results: cryptography.io, PyCrypto and (to some extent) PyNaCl performed best on these metrics. The security results, however, were somewhat different. PyCrypto and M2Crypto were worst, while Keyczar performed best. PyNaCl also had strong security results; cryptography.io exhibited strong security for the symmetric tasks but poor security for asymmetric tasks. These results suggest that the relationship between "usable" design, developer satisfaction, and security outcomes is a complex one.

A. Participants

In total, we sent 52 448 email invitations. Of these, 5 918 (11.3%) bounced, and another 698 (1.3%) requested to be removed from our list, a request we honored.

A total of 1 571 people agreed to our consent form; 660 (42.0%) dropped out without taking any action, most likely because the initial task seemed too difficult or time-consuming. The other 911 proceeded through at least one task; of these, 337 proceeded to the exit survey, and 282 completed it with valid responses.[1] Of these, 26 were excluded for failing to use their assigned library. Unless otherwise noted, we report results for the remaining 256 participants, who proceeded through all tasks, used their assigned library, and completed the exit survey with valid responses.

An additional 61 participants attempted to reach the study but encountered technical errors in our infrastructure, mainly due to occasional AWS pool exhaustion during times of high demand.

Our 256 participants reported ages between 18 and 63 (mean 29.4, sd 7.9), and the vast majority of them reported being male (238, 93.0%).

We successfully reached the professional developer demographic we targeted. Almost all (247, 96.5%) had been programming in general for more than two years, and 81.2% (208) had been programming in Python for more than two years. Most participants (196, 76.6%) reported programming as (part of) their primary job; of those, 147 (75.0%) used Python in their primary job. Most participants (195, 76.2%) said they had no IT-security background.

While the developers we invited represent a random sample from GitHub, our valid participants are a small, self-selected subset. Table IV and Figure 2 detail available GitHub demographics for both groups. Our participants appear to be slightly more active on GitHub than average: owning more public repositories, having more followers, having older accounts, and being more likely to provide optional profile information. This may correspond to their self-reported high levels of programming experience and professional status.

B. Regression models

In the following subsections, we apply regression models to analyze our results in detail. To analyze binary outcomes (e.g., secure vs. insecure), we use logistic regression; to analyze numeric outcomes (e.g., SUS score), we use linear regression. When we consider results on a per-task rather than a per-participant basis (for security and functionality results, as well as perceived security), we use a mixed model that adds a random intercept to account for multiple tasks from the same participant.

[1] We define invalid responses as providing straight-line answers to all questions or writing off-topic or abusive comments in free-text responses.

	Invited	Valid
Hireable	19.5%	37.9%
Company listed	28.0%	42.2%
URL to Blog	34.7%	55.6%
Biography added	8.1%	16.3%
Location provided	49.9%	75.8%
Public gists (median)	0	1
Public repositories (median)	12	20
Following (users, median)	1	2
Followers (users, median)	3	7
GitHub profile creation (days ago, median)	1 415	1 589
GitHub profile last update (days ago, median)	50	38

TABLE IV
GitHub demographics for the 50 000 invited users and for our 256 valid participants.

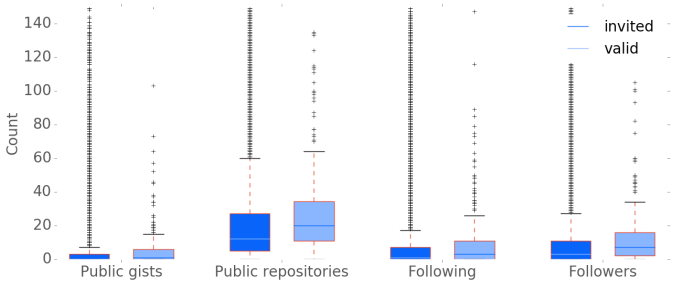


Fig. 2. Boxplots comparing our invited participants (a random sample from GitHub) with those who provided valid participation. The center line indicates the median; the boxes indicate the first and third quartiles. The whiskers extend to ±1.5 times the interquartile range. Outliers greater than 150 were truncated for space.

For each regression analysis, we consider a set of candidate models and select the model with the lowest Akaike Information Criterion (AIC) [67]. The included factors are described in Table V. We consider candidate models consisting of the required factors *library* and *encryption mode*, as well as (where applicable) the participant random intercept, plus every possible combination of the optional variables.

We report the outcome of our regressions in tables. Each row measures change in the analyzed outcome related to changing from the *baseline* value for a given factor to a different value for that factor (e.g., changing from asymmetric to symmetric encryption). Logistic regressions produce an odds ratio (O.R.) that measures change in likelihood of the targeted outcome; baseline factors by construction have O.R.=1. For example, Table VII indicates that M2Crypto participants were 0.55× as likely to complete all tasks as participants in the baseline PyCrypto condition. In constrast, linear regressions measure change in the absolute value of the outcome; baseline factors by construction have coef=0. In each row, we also provide a 95% confidence interval (C.I.) and a p-value indicating statistical significance.

For each regression, we set the library PyCrypto as the baseline, as it has the most download counts of all libraries we included in our study, and can therefore be considered as the most common "default" crypto library for Python. In addition, we used the set of symmetric tasks as the baseline, as these correspond to the simpler and more basic form of encryption. All baseline values are given in Table V.

C. Dropouts

We first examine how library and encryption mode affected participants' dropout rates, as we believe that dropping out of the survey is a first (if crude and oversimplified) measure of how much effort was required to solve the assigned tasks with the assigned library. Table VI details how many participants in each condition reached each stage of the study.

We test whether library and encryption mode affect dropout rate using a logistic regression model (see Section IV-B) examining whether each participant who consented proceeded through all tasks and started the exit survey. (We use the start of the survey here because dropping out at the survey stage seems orthogonal to library type.) For this model, we include only the library-encryption mode interactions as an optional factor, because we do not have experience or security background data for the participants who dropped out.

The final model (see Table VII) indicates that asymmetric-encryption participants were only about half as likely to proceed through all tasks as participants assigned to symmetric encryption, which was statistically significant. Compared to the "default" choice of PyCrypto, participants assigned to M2Crypto and Keyczar were about half as likely to proceed through all tasks, which was also statistically significant. PyNaCl exhibits a higher dropout rate than PyCrypto; however, this trend was not significant. cryptography.io matches PyCrypto's dropout rate. Although the two-way interactions are included in the final model, none exhibits a significant result.

Overall, these results suggest that PyCrypto (approximate default) and cryptography.io (designed for usability, with relatively complete documentation) were least likely to drive participants away. Keyczar, also designed for usability, performed worst on this metric.

D. Functionality results

We next discuss the extent to which participants were able to produce functional solutions—that is, solutions that produced a key or encrypted and decrypted some content without generating an exception.[2] We observed a wide variance in functional results across libraries and encryption types, ranging from asymmetric Keyczar (13.7% functional) to symmetric cryptography.io and symmetric PyNaCl (89.5% and 87.9% functional respectively). Figure 3 illustrates these results.

To examine these results more precisely, we applied a logistic regression, as described in Section IV-B, to model the factors that affect whether or not each individual task was marked as functional. The final model (see Table VIII) shows that M2Crypto and Keyczar are significantly worse for functionality than the baseline PyCrypto; cryptography.io and PyNaCl appear slightly better, but the difference is not

[2] Participants who skipped a task are counted as functionally incorrect for that task.

Factor	Description	Baseline
Required factors		
Library	The cryptographic library used.	PyCrypto
Encryption mode	Asymmetric or Symmetric	Symmetric
Optional factors		
Experienced	True if a programming in Python is part of participant's job, and/or if participant has been programming in Python for more than five years; otherwise false. Self-reported.	False
Security background	True or false, self-reported.	False
Library experience	Whether the participant has used the library before, seen code that used it but not used it themselves; or neither. Self-reported.	No experience
Copy-paste	Whether the participant pasted code during this task. Measured, per-task regressions only.	False
Library × Mode	Interaction between the library and encryption mode factors described above.	cryptography.io :asymmetric

TABLE V
Factors used in regression models. Categorical factors are individually compared to the baseline. Final models were selected by minimum AIC; candidates were defined using all possible combinations of optional factors, with both required factors included in every candidate.

Library	Mode	Consented	Started Survey	Total Valid
PyCrypto	sym	136	48	41
	asym	175	37	24
M2Crypto	sym	157	36	20
	asym	174	35	27
cryptography.io	sym	136	48	39
	asym	174	22	19
Keyczar	sym	136	26	20
	asym	173	24	17
PyNaCl	sym	136	34	29
	asym	174	27	20
Total		1 571	337	256

TABLE VI
The number of participants who progressed through each phase of the study, by condition. Each column is a subset of the previous columns.

Factor	O.R.	C.I.	*p*-value
M2Crypto	0.55	[0.33, 0.91]	0.02*
cryptography.io	1.00	[0.61, 1.64]	1
Keyczar	0.43	[0.25, 0.75]	0.003*
PyNaCl	0.61	[0.36, 1.03]	0.065
asymmetric	0.49	[0.3, 0.81]	0.006*
M2Crypto:asymmetric	1.72	[0.83, 3.57]	0.144
cryptography.io:asymmetric	0.54	[0.25, 1.16]	0.112
Keyczar:asymmetric	1.39	[0.63, 3.05]	0.418
PyNaCl:asymmetric	1.12	[0.53, 2.39]	0.768

TABLE VII
Results of the final logistic regression model examining whether participants who consented proceeded through all tasks and continued to the survey. Odds ratios (O.R.) indicate relative likelihood of continuing. Statistically significant factors indicated with *. See Section IV-B for further details.

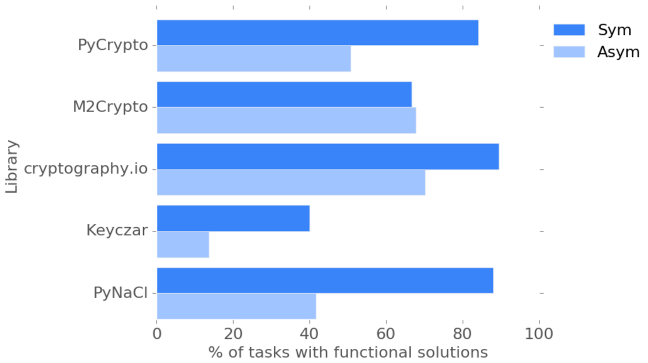


Fig. 3. Percentage of tasks for which participants generated functional solutions, by condition.

significant. Most notably, Keyczar is estimated as only 10% as likely to produce a functional result. By comparing confidence intervals, we see that Keyczar is also significantly worse than PyNaCl and cryptography.io. The results also show that symmetric tasks were about 6× (0.16^{-1}) as likely as asymmetric tasks to have functional solutions, and that using code generated via copy-and-paste improves a task's odds of functionality about 3× (both significant). The participant's Python experience level, security background, and experience with their assigned library do not appear in the final model, suggesting they are not significant factors in the functionality results.

In general, the set of asymmetric cryptography tasks was harder to solve in a functionally correct way than the set of symmetric cryptography tasks. This seem to be largely because we included X.509 certificate handling in the set of asymmetric cryptography tasks. Two of the libraries specifically designed to be easy to use (Keyczar and PyNaCl) do not support X.509 certificate handling out of the box, so these tasks had to be done via workarounds or could not be solved at all. On the other hand, the low-level X.509 certificate APIs of M2Crypto and PyCrypto require developers to deal with many cryptographic details (e.g., root certificate stores and certificate details such as the Common Name or Subject Alternative Name), which might have an impact on

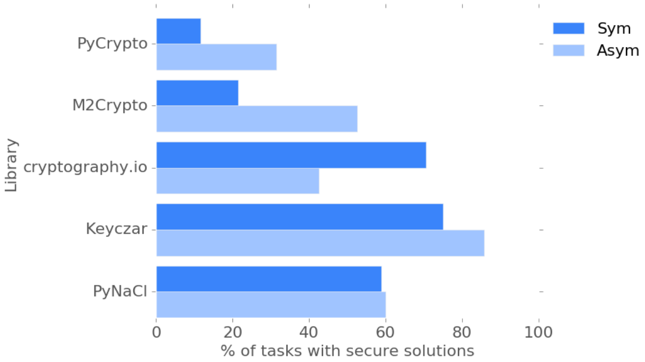


Fig. 4. Percentage of tasks with secure solutions, considering only tasks with functional solutions, by condition.

functionality in addition to security.

The only significant interaction in the final model is between M2Crypto and asymmetric tasks: these tasks were about 8× more likely than expected to be marked functional. Indeed, M2Crypto is the only library (see Figure 3) for which symmetric tasks were (slightly) less functional than asymmetric tasks. We hypothesize that this is caused by the requirement that developers have to choose many cryptographic details for both symmetric and asymmetric encryption in M2Crypto.

Factor	**O.R.**	**C.I.**	***p*-value**
M2Crypto	0.26	[0.09, 0.69]	0.007*
cryptography.io	1.68	[0.61, 4.61]	0.311
Keyczar	0.10	[0.04, 0.26]	$<$ 0.001*
PyNaCl	1.58	[0.55, 4.56]	0.394
asymmetric	0.16	[0.07, 0.38]	$<$ 0.001*
copy-paste	3.29	[1.97, 5.49]	$<$ 0.001*
M2Crypto:asymmetric	8.14	[2.29, 28.95]	0.001*
cryptography.io:asymmetric	1.53	[0.4, 5.75]	0.532
Keyczar:asymmetric	1.50	[0.36, 6.22]	0.578
PyNaCl:asymmetric	0.49	[0.13, 1.86]	0.293

TABLE VIII
Results of the final logistic regression mixed model examining which factors correlate with task functionality. Odds ratios indicate relative likelihood of a task being functionally correct. Statistically significant values indicated with *. See Section IV-B for further details.

Factor	**O.R.**	**C.I.**	***p*-value**
M2Crypto	2.20	[0.68, 7.11]	0.186
cryptography.io	19.34	[7.78, 48.03]	<0.001*
Keyczar	24.54	[6.31, 95.43]	<0.001*
PyNaCl	11.29	[4.46, 28.61]	<0.001*
asymmetric	3.58	[1.28, 10.03]	0.015*
sec. bkgrd.	1.57	[0.94, 2.61]	0.083
M2Crypto:asymmetric	1.09	[0.25, 4.73]	0.909
cryptography.io:asymmetric	0.08	[0.02, 0.31]	<0.001*
Keyczar:asymmetric	0.54	[0.04, 7.37]	0.642
PyNaCl:asymmetric	0.29	[0.07, 1.2]	0.088

TABLE IX
Results of the final logistic regression mixed model examining which factors correlate with task security, among only tasks that were functional. Odds ratios indicate relative likelihood of a solution being secure. Statistically significant values indicated with *. See Section IV-B for further details.

E. Security results

Next, we consider whether participants whose code was functional also produced secure solutions. As with functionality, we observed a broad range of results (see Figure 4). Overall, Keyczar was notably secure (for a small sample) and PyCrypto and to a lesser extent M2Crypto were notably insecure.

We again apply logistic regression (Section IV-B) to investigate the factors that influence security; we include only functional task solutions in this analysis. The results are shown in Table IX. The final model shows that compared to the baseline PyCrypto, every library appears to produce better security; all of these except M2Crypto are significant. At the extreme, Keyczar is estimated almost 25× as likely to produce a secure solution. This is particularly notable because Keyczar was so difficult: only 16 and seven participant tasks, respectively, exhibited functional symmetric and asymmetric solutions, but 12 and six of these respectively were secure, the highest per-capita of any library. The regression results also show that at baseline, asymmetric tasks were about 3× more likely to exhibit secure code than symmetric tasks. The final model also indicates that tasks from participants with a security background were about 1.5× more likely to be secure; Python experience level and experience directly with the assigned library do not seem to affect security noticeably, as they do not appear in the final model. The only significant interaction term is between cryptography.io and asymmetric: cryptography.io is the only library for which asymmetric performed less securely. We hypothesize that this is because the symmetric tasks could be completed using the library's high-level "recipes" layer, while the asymmetric tasks required the participant to work with the low-level "hazmat" layer.

Security perception. In the exit survey, we showed participants the code they had written to solve each task and asked them (on a five-point Likert scale from Strongly Agree to Strongly Disagree) whether they thought their solution was secure. We did not define security, as we wanted to know whether our participants were satisfied with the security properties of their code in general, rather than meeting a specific threat model. Across all libraries, the majority of our participants were convinced that their solution was secure. The median (excluding 10% of tasks for which participants answered "I don't know") was no lower than "neutral" across all combinations of libraries and encryption modes; security confidence was highest for cryptography.io and PyNaCl (both encryption modes), as well as PyCrypto and Keyczar (asymmetric), all of which had median value "agree."

In considering these answers, we are most interested in tasks for which we rated the solution insecure, but the participant agreed or strongly agreed that their solution for that task

Factor	O.R.	C.I.	*p*-value
M2Crypto	0.59	[0.25, 1.38]	0.221
cryptography.io	0.58	[0.27, 1.27]	0.176
Keyczar	0.25	[0.05, 1.3]	0.099
PyNaCl	0.62	[0.27, 1.46]	0.277
asymmetric	1.32	[0.72, 2.42]	0.373
sec. bkgrd.	1.65	[0.86, 3.14]	0.13

TABLE X
Results of the final logistic regression mixed model examining factors correlating with erroneous belief that a task is secure. Odds ratios indicate relative likelihood of this belief. Some trends are observable, but no results are statistically significant. See Section IV-B for further details.

Library	Mode	Mean SUS	Mean API Scale
PyCrypto	sym	63.9	64.2
	asym	47.8	52.5
M2Crypto	sym	33.9	32.5
	asym	36.4	35.6
cryptography.io	sym	67.2	67.7
	asym	52.3	61.6
Keyczar	sym	40.8	40.9
	asym	32.5	26.9
PyNaCl	sym	67.2	66.8
	asym	59.5	57.1

TABLE XI
Mean SUS scores and scores on our new API usability scale, by condition.

was secure. These situations are potentially dangerous, as the developer mistakenly believes they have achieved security. Overall, 78 of 396 tasks (19.7%) fell into this category, a disappointingly high number. To examine factors that correlate with this situation, we applied a mixed-model logistic regression, as described in Section IV-B, with outcome *dangerous error* or *not* per task. The results are shown in Table X. Although some trends are observable, the final model finds no significant results; this suggests that at least at this sample size, no particular factors were significantly associated with a higher likelihood of erroneous belief.

F. Participant opinions

Our self-reported usability metrics reveal large differences between the libraries. Table XI lists the average SUS scores by condition. Overall, PyNaCl and cryptography.io performed best, while M2Crypto and Keyczar performed worst. Overall, these SUS scores are quite low; a score of 68 is considered average for end-user products and systems [63], and even our best-performing condition does not reach this standard. This suggests that even the most usable libraries we tested have considerable room for improvement.

Using a linear regression model (see Section IV-B), we analyzed the impact of library and encryption mode, shown in Table XII. We find that M2Crypto and Keyczar are significantly less usable than the baseline PyCrypto; PyNaCl is significantly more usable. Unsurprisingly, symmetric-condition participants reported significantly more usability than asymmetric-condition participants. The final model indicates that security background and having seen the assigned library before were both associated with a significant increase in usability. Having used the library before was associated with an increase relative to no familiarity, but this trend was not significant, probably because of the very small sample size: only 18 participants reported having used their assigned library before. Python experience was included in the final model but was not a signficiant covariate; the final model did not include any interactions between library and encryption mode.

We compiled our additional usability questions, drawn from prior work as described in Section III-G, into a score out of 100 points. The results were similar to the SUS, and in fact, the two scores were significantly correlated (Kendall's τ=0.65, $p < 0.001$). Using Cronbach's alpha, we determined that the scale's internal reliability was high ($\alpha = 0.98$).

Factor	Coef.	C.I.	*p*-value
M2Crypto	-20.57	[-27.62, -13.52]	<0.001*
cryptography.io	5.04	[-1.52, 11.61]	0.131
Keyczar	-18.07	[-25.85, -10.3]	<0.001*
PyNaCl	7.56	[0.48, 14.64]	0.036*
asymmetric	-9.60	[-14.13, -5.08]	<0.001*
experienced	3.79	[-1.33, 8.91]	0.146
sec. bkgrd.	6.22	[0.98, 11.46]	0.02*
seen lib	6.62	[0.39, 12.85]	0.037*
used lib	3.33	[-5.95, 12.6]	0.481

TABLE XII
Linear regression model examining SUS scores. The coefficient indicates the average difference in score between the listed factor and the base case. Significant values indicated with *. $R^2 = 0.376$. See Section IV-B for further details.

Table XIII shows the results of a linear regression examining score on our scale. As before, M2Crypto and Keyczar are significantly worse than PyCrypto. Using this measure, cryptography.io is significantly better than PyCrypto, while PyNaCl is better than PyCrypto but not significantly so. Also as before, significantly higher scores were correlated with symmetric tasks and with having seen the assigned library before. Having used the library before was again correlated with higher scores, but not significantly so, probably due to sample size. Security background was included in the final model but not significant; Python experience and interactions between library and encryption mode were not included in the final model.

The answers to questions about the API documentation indicate that Keyczar and M2Crypto have a sizable problem with their documentation: Our participants consistently answered that they found neither helpful explanations nor helpful code examples in the documentation, and that they had to spend a lot of time reading the documentation before they could solve the tasks. Altogether, they found the documentation for Keyczar and M2Crypto not helpful. This corresponded to responses saying that the tasks were not straightforward to implement

Factor	Coef.	C.I.	*p*-value
M2Crypto	-22.44	[-28.54, -16.35]	<0.001*
cryptography.io	7.21	[1.45, 12.97]	0.014*
Keyczar	-21.59	[-28.41, -14.77]	<0.001*
PyNaCl	5.66	[-0.5, 11.82]	0.072
asymmetric	-8.00	[-11.99, -4.02]	<0.001*
sec. bkgrd.	3.94	[-0.66, 8.54]	0.093
seen lib	6.60	[1.12, 12.09]	0.019*
used lib	6.74	[-1.41, 14.88]	0.104

TABLE XIII
Linear regression model examining scores on our cognitive-dimension-based scale. The coefficient indicates the average difference in score between the listed factor and the base case (PyCrypto and symmetric, respectively). Significant values indicated with *. $R^2 = 0.466$. See Section IV-B for further details.

for these two libraries. Interestingly, for cryptography.io, the perceived effort that had to be invested into understanding the library in order to be able to work on the tasks was the lowest. For cryptography.io, PyNaCl, and PyCrypto, the developers felt that after having used the library to solve the tasks, they had a pretty good understanding of how the library worked.

For color, we include a few exemplar quotes from our participants who chose to comment on the documentation. One participant said the Keyczar documentation was "awful and doesn't seem to document its Python API at all." A second said, "I don't understand why you have an API with no search feature and functional descriptions. This is insane," and a third commented that "The linked document is so unkind that I must read the code." A third Keyczar participant left an ASCII-art comment spelling out "Your documentation is bad and you should feel bad."

One participant assigned to M2Crypto called the documentation "solidly awful," "just terrible," and "completely unusable." The same participant inquired whether our request to use this library was "a joke" or "part of the study." Other M2Crypto participants said "the linked documentation is wildly insufficient" and M2Crypto's "interface is arcane and documentation hard to understand." Several participants assigned to this library commented that they had to revert to Stack Overflow posts or blog entries found via search engines to be able to work on the tasks at all.

In contrast, one participant working with cryptography.io called a tutorial contained in the documentation "amazing!" while stating that "The comparable OpenSSL docs make one want to jump off a cliff." Another said the documentation "was confusing at first, but later I got the hang of it."

G. Examining individual tasks

Success in solving the tasks varied not only across libraries, but also across individual tasks, as illustrated in Figure 5. We analyze these results for trends, rather than statistical significance, to avoid diluting our statistical power by testing the same results in multiple ways.

Encryption proved easiest. Symmetric participants achieved 85.2% functional success, with 70.1% of those rated secure; 72.0% of asymmetric encryption tasks were functional, with 78.8% of those rated secure. In contrast, the hardest task to solve overall dealt with certificate validation. Only 22.4% of asymmetric participants were able to provide a functional solution, and not a single one was secure. Key generation tasks fell in the middle.

Investigating security errors. We also examined trends in the types of security errors made by our participants. (For a full accounting, see Table XIV in Appendix B.)

We first consider symmetric cryptography, and in particular situations where participants were allowed to make security choices. Only M2Crypto and PyCrypto allow developers to choose an encryption algorithm; interestingly, all 11 PyCrypto participants selected DES (insecure), but no M2Crypto participants chose an insecure algorithm. While M2Crypto's official API documentation does not provide code examples, the first results on Google when searching "m2crypto encryption" provide code snippets that use AES. The PyCrypto documentation does provide code examples for symmetric encryption and discourages the use of DES as a weak encryption algorithm. However, the first Google results when searching "pycrypto encryption" provide code examples that use DES. Nine of the 11 participants who used DES mentioned specific blog posts and Stack Overflow posts that we later determined to have insecure code snippets.

Similarly, allowing developers to pick modes of operation resulted in relatively many vulnerabilities. PyCrypto participants chose the insecure ECB as mode of operation explicitly or did not provide a mode of operation parameter at all (ECB is the default). As with selecting an encryption algorithm, affected participants reported using blog posts and Stack Overflow posts containing insecure snippets as information sources. PyCrypto participants chose static IVs more frequently than those using other libraries; interestingly, this corresponds to not mentioning the importance of a truly random IV in the documentation. Relatedly, requiring developers to pick key sizes manually frequently resulted in too-small keys, across libraries.

Interestingly, PyCrypto participants were most likely to fail to use any key derivation function, possibly because the documentation uses a plain string for an encryption key. PyNaCl and PyCrypto participants used an insecure custom key derivation function more frequently than participants in other conditions: they frequently used a simple hash function for key stretching. cryptography.io participants, in contrast, performed exceedingly well on this task, likely because the included PBKDF2 function is well documented and close to the symmetric encryption example. On the negative side, cryptography.io users picked static salts for PBKDF2 more frequently than others, even though the code example in the API documentation uses a random salt; however, no explanation on the importance of using a random value is given. Storing encryption keys in plaintext rather than encrypted was also common across all libraries.

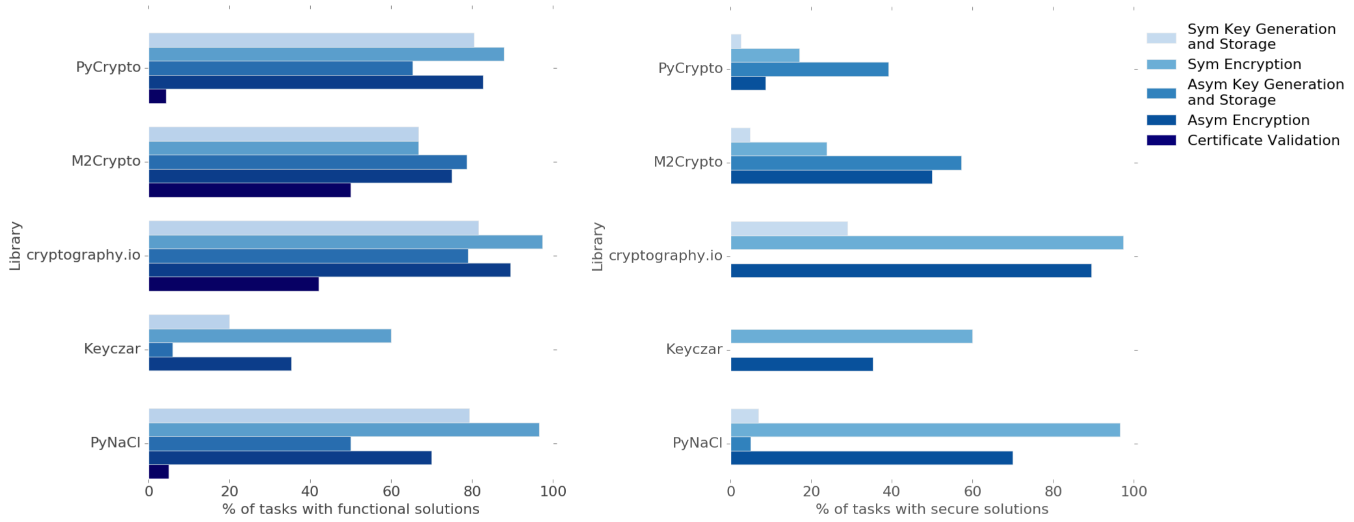


Fig. 5. Percentage of tasks with functionally correct solutions (left), and percentage of functional solutions that were rated secure (right), organized by library and task type.

Generating and storing asymmetric keys was significantly less vulnerable to weak cryptographic choices. Only PyCrypto and M2Crypto participants failed to pick sufficiently secure RSA key sizes, potentially due again to insecure code examples (mentioning 1024-bit keys) among the top Google search results. Since all libraries but Keyczar and PyNaCl provide a private-key export function that offers encryption, asymmetric private-key storage had comparably few insecurities. However, PyNaCl users had to manually encrypt their private key and ran into similar security problems as the symmetric-encryption users mentioned above. Asymmetric encryption produced relatively few security errors.

Certificate validation was the most challenging task. Across all libraries, participants had trouble properly implementing signature validation, hostname verification, CA checks, and validity checks. This may be caused by task complexity and insufficient API support.

V. Discussion and Conclusion

Our results suggest that usability and security are deeply interconnected in sometimes surprising ways. We distill some high-level findings derived from our individual results and suggest future directions for library design and further research.

Simplicity does promote security (to a point). In general, the simplified libraries we tested produced more secure results than the comprehensive libraries, validating the belief that simplicity is better. Further, cryptography.io proved secure for the symmetric tasks (primarily doable via the simplified “recipes” layer) but not for the asymmetric tasks (primarily requiring use of the complex “hazmat” layer). This reinforces both the idea that simplicity promotes security and the need for simplified libraries to offer a broader range of features.

However, even simplified libraries did not entirely solve the security problem; in all but one condition, the rate of security success was below 80%. These security errors were frequently caused by missing features (discussed next). Worse, for 20% of functional solutions, the participant rated their code as secure when it was not; this indicates a dangerous gap in recognition of potential security problems.

Features and documentation matter for security. Several of the libraries we selected did not (or not well) support tasks auxiliary to encryption and decryption, such as secure key storage and password-based key generation. These missing features caused many of the insecure results in the otherwise-successful simplified libraries. We argue that to be usably secure, a cryptographic API must support such auxiliary tasks, rather than relying on the developer to recognize the potential for danger and identify a secure alternate solution. Further, we suggest that cryptographic APIs should be designed to support a reasonably broad range of use cases; requiring developers to learn and use new APIs for closely related tasks seems likely to drive them back to comprehensive libraries like PyCrypto or M2Crypto, which pose security risks.

Documentation is also critical. PyCrypto, for example, contains symmetric encryption examples that use AES in ECB mode, which is prima facie insecure. Participants who left the PyCrypto documentation to search for help on Stack Overflow and blogs often ended up with insecure solutions; this suggests the importance of creating official documentation that is useful enough to keep developers from searching out unvetted, potentially insecure alternatives. Many participants copied these examples in their solutions. In contrast, the excellent code examples for PyNaCl and in the cryptography.io “recipes” layer appear to have contributed to high rates of security success.

What do we mean by usable? Despite claims of usability and a simplified API, Keyczar proved the most difficult to use of our chosen libraries. This was caused primarily by two issues: poor documentation (as measured by our API usability scale) and the lack of documented support for key generation in code, rather than requiring interaction at the command line. Those few participants who successfully achieved functional code had very high rates of security, but in practice developers who give up on a library because they cannot make it work for the desired task will not be able to take advantage of potential security benefits. For example, developers who have difficulty with Keyczar might turn to PyCrypto, which participants preferred but which showed poor security results.

A blueprint for future libraries. Taken together, our results suggest several important considerations for designers of future cryptographic libraries. First, the recent emphasis on simplifying APIs (and choosing secure defaults) has provided improvement; we endorse continuing in this direction. We suggest, however, that library designers go further, by treating documentation quality as a first-class requirement, with particular emphasis on secure code examples. We also recommend that library designers consider a broad range of potential tasks users might need to accomplish cryptographic goals, and build support for each of them into a more comprehensive whole.

Our results suggest that supporting holistic, application-level tasks with ready-to-use APIs is the best option. That said, we acknowledge that it may be difficult or impossible to predict all tasks API users may want or need. Therefore, where lower-level features are necessary, they should be intentionally designed to make combining them into more complex tasks securely as easy as possible.

Looking forward, further research is needed to design and evaluate libraries that meet these goals. Some changes can also be made within existing libraries—for example, improving documentation, changing insecure defaults to secure defaults, or even adding compiletime or runtime warnings for insecure parameters. These changes should be evaluated involving future users both before they are deployed and longitudinally to see how they affect outcomes within real-world code. We also hope to refine and expand the usability scale developed in this paper to create an evaluation framework for security APIs generally, providing both feedback and guidance for improvement.

VI. Acknowledgments

The authors would like to thank Mary Theofanos, Julie Haney, Jason Suagee, and the anonymous reviewers for providing feedback; Marius Steffens and Birk Blechschmidt for helping to test the infrastructure; Matt Bradley and Andrea Dragan for help managing multi-institution ethics approvals; and all of our participants for their contributions. This work was supported in part by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0344,16KIS0656), and by the U.S. Department of Commerce, National Institute for Standards and Technology, under Cooperative Agreement 70NANB15H330.

References

[1] Amnesty International USA, “Encryption - A Matter of Human Rights,” 2016. [Online]. Available: https://www.amnestyusa.org/sites/default/files/encryption_-_a_matter_of_human_rights_-_pol_40-3682-2016.pdf

[2] R. J. Anderson, “Why cryptosystems fail,” *Communications of the ACM*, vol. 37, 1994.

[3] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, “The most dangerous code in the world: validating SSL certificates in non-browser software,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012).* ACM, 2012.

[4] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. R. Butler, “Mo(bile) money, mo(bile) problems: analysis of branchless banking applications in the developing world,” in *Proceedings of the 24th USENIX Security Symposium (USENIX Security 2015).* USENIX Association, 2015.

[5] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An empirical study of cryptographic misuse in Android applications,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013).* ACM, 2013.

[6] S. Fahl, M. Harbach, T. Muders, M. Smith, and U. Sander, “Helping Johnny 2.0 to encrypt his Facebook conversations,” in *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS 2012).* ACM, 2012.

[7] J. Viega, M. Messier, and P. Chandra, *Network Security with OpenSSL.* O’Reilly Media, 2002.

[8] “Cryptography.io.” [Online]. Available: https://cryptography.io

[9] D. J. Bernstein, T. Lange, and P. Schwabe, “The security impact of a new cryptographic library,” in *Proceedings of the 2nd International Conference on Cryptology and Information Security in Latin America (LATINCRYPT 2012).* Springer-Verlag, 2012.

[10] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, “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 2012).* ACM, 2012.

[11] L. Onwuzurike and E. De Cristofaro, “Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps,” *arXiv.org*, 2015.

[12] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl, “To pin or not to pin—helping app developers bullet proof their tls connections,” in *Proceedings of the 24th USENIX Security Symposium (USENIX Security 2015).* USENIX Association, 2015.

[13] H. Perl, S. Fahl, and M. Smith, “You won’t be needing these any more: On removing unused certificates from trust stores,” in *Proceedings of 18th International Conference on Financial Cryptography and Data Security (FC 2014).* Springer Berlin Heidelberg, 2014.

[14] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, “SoK: Lessons Learned from Android Security Research for Appified Software Platforms,” in *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP 2016)*, 2016.

[15] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, “Hey, you, get off of my clipboard,” in *Proceedings on Financial Cryptography and Data Security (FC 2013).* Springer, 2013.

[16] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, “Rethinking SSL development in an appified world,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013).* ACM, 2013.

[17] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, “Why does cryptographic software fail?” in *Proceedings of the 5th Asia-Pacific Workshop on Systems.* ACM, 2014.

[18] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, ““Jumping Through Hoops”: Why do Java Developers Struggle With Cryptography APIs?” in *Proceedings of the 37th International Conference on Software Engineering (ICSE 2016)*, 2016.

[19] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, “You Get Where You’re Looking For: The Impact of Information Sources on Code Security,” in *Proceedings of the 37th IEEE Symposium on Symposium on Security and Privacy (SP 2016)*, 2016.

[20] S. Arzt, S. Nadi, K. Ali, E. Bodden, and S. Erdweg, “Towards secure integration of cryptographic software,” in *Proceedigs of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2015)*, 2015.

[21] S. Indela, M. Kulkarni, K. Nayak, and T. Dumitra, “Helping Johnny encrypt: Toward semantic interfaces for cryptographic frameworks,” in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*, 2016.

[22] B. A. Myers and J. Stylos, “Improving API usability,” *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, 2016.

[23] J. Nielsen, *Usability engineering*. Morgan Kaufmann, 1993.

[24] S. Clarke, "Using the cognitive dimensions framework to design usable APIs," https://blogs.msdn.microsoft.com/stevencl/2003/11/14/using-the-cognitive-dimensions-framework-to-design-usable-apis/.

[25] J. Bloch, "How to design a good API and why it matters," in *Companion to the 21st ACM SIGPLAN Conference*. ACM, 2006.

[26] M. Henning, "API design matters," *Queue*, vol. 5, no. 4, pp. 24–36, 2007.

[27] M. Green and M. Smith, "Developers are Not the Enemy!: The Need for Usable Security APIs," *IEEE Security & Privacy*, vol. 14, no. 5, pp. 40–46, 2016.

[28] P. Gorski and L. L. Iacono, "Towards the usability evaluation of security apis," in *Proceedings of the Tenth International Symposium on Human Aspects of Information Security & Assurance (HAISA 2016)*, 2016.

[29] C. Wijayarathna, N. A. G. Arachchilage, and J. Slay, "Generic cognitive dimensions questionnaire to evaluate the usability of security apis," in *Proceedings of the 19th International Conference on Human-Computer Interaction (to appear)*, 2017.

[30] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang, "It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[31] G. Wurster and P. C. van Oorschot, "The developer is the enemy," in *Proceedings of the 2008 New Security Paradigms Workshop (NSPW 2008)*. ACM, 2008.

[32] M. Finifter and D. Wagner, "Exploring the relationship between web application development tools and security," in *Proceedings of the 2nd USENIX conference on Web application development (WebApps 2011)*, 2011.

[33] L. Prechelt, "Plat_forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 95–108, 2011.

[34] T. Scheller and E. Kühn, "Usability Evaluation of Configuration-Based API Design Concepts," in *Human Factors in Computing and Informatics*. Springer Berlin Heidelberg, 2013, pp. 54–73.

[35] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium*. ACM, 2008.

[36] B. Ellis, J. Stylos, and B. Myers, "The Factory Pattern in API Design: A Usability Evaluation," in *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE, 2007.

[37] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013.

[38] C. Burns, J. Ferreira, T. D. Hellmann, and F. Maurer, "Usable results from the field of API usability: A systematic mapping and further analysis," in *Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing* , 2012.

[39] "GitHut: A Small Place to discover languages in GitHub," 2016. [Online]. Available: http://githut.info

[40] S. Willden, "Keyczar Design Philosophy," 2015. [Online]. Available: https://github.com/google/keyczar/wiki/KeyczarPhilosophy

[41] "OpenSSL." [Online]. Available: https://www.openssl.org/

[42] "PyCrypto." [Online]. Available: https://www.dlitz.net/software/pycrypto

[43] "M2Crypto." [Online]. Available: https://pypi.python.org/pypi/M2Crypto

[44] "Keyczar." [Online]. Available: https://github.com/google/keyczar

[45] "PyNaCl." [Online]. Available: https://pynacl.readthedocs.io/en/latest

[46] "pyOpenSSL." [Online]. Available: http://www.pyopenssl.org/en/stable

[47] "tlslite." [Online]. Available: http://trevp.net/tlslite/

[48] "bcrypt." [Online]. Available: https://github.com/pyca/bcrypt

[49] "gnupg." [Online]. Available: https://github.com/isislovecruft/python-gnupg

[50] "pycryptopp." [Online]. Available: https://tahoe-lafs.org/trac/pycryptopp

[51] "scrypt." [Online]. Available: http://bitbucket.org/mhallin/py-scrypt

[52] "simple-crypt." [Online]. Available: https://github.com/andrewcooke/simple-crypt

[53] "pysodium." [Online]. Available: https://github.com/stef/pysodium

[54] "ed25519." [Online]. Available: https://pypi.python.org/pypi/ed25519

[55] "pyaes." [Online]. Available: https://github.com/ricmoo/pyaes

[56] "PyCryptodome." [Online]. Available: http://pycryptodome.readthedocs.io

[57] "PyMe." [Online]. Available: http://pyme.sourceforge.net

[58] "pyDes." [Online]. Available: https://github.com/toddw-as/pyDes

[59] "tls." [Online]. Available: https://github.com/pyca/tls

[60] "GitHub Archive." [Online]. Available: https://www.githubarchive.org

[61] "Jupyter notebook." [Online]. Available: http://jupyter.org/

[62] "The Sodium crypto library (libsodium)." [Online]. Available: https://libsodium.org

[63] P. W. Jordan, B. Thomas, B. A. Weerdmeester, and A. L. McClelland, "SUS: A "quick and dirty" usability scale," in *Usability Evaluation in Industry*. Taylor and Francis, 1996, pp. 189–194.

[64] National Institute of Standards and Technology (NIST), "NIST Special Publication 800-57 Part 1 Revision 4: Recommendation for Key Management," 2016. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

[65] S. Josefsson, "PKCS #5: Password-Based Key Derivation Function 2 (PBKDF2) Test Vectors," RFC 6070, 2011.

[66] National Institute of Standards and Technology (NIST), "NIST Special Publication 800-63B Digital Authentication Guideline," 2016. [Online]. Available: https://pages.nist.gov/800-63-3/sp800-63b.html

[67] K. P. Burnham, "Multimodel Inference: Understanding AIC and BIC in Model Selection," *Sociological Methods & Research*, vol. 33, no. 2, pp. 261–304, 2004.

Appendix

A. Exit survey questions

Task-specific questions: Asked about each task

Please rate your agreement to the following statements: (Strongly agree; agree; neutral; disagree; strongly disagree; I don't know.)

- I think I solved this task correctly.
- I think I solved this task securely.
- The documentation was helpful in solving this task.

General questions

- Are you aware of a specific library or other resource you would have preferred to solve the tasks? Which? (Yes with free response; no; I don't know.)
- Have you used or seen the assigned library before? For example, maybe you worked on a project that used the assigned library, but someone else wrote that portion of the code. (I have used the assigned library before; I have seen the assigned library used but have not used it myself; No, neither; I dont know.)
- Have you written or seen code for tasks similar to this one before? For example, maybe you worked on a project that included a similar task, but someone else wrote that portion of the code. (I have written similar code; I have seen similar code but have not written it myself; No, neither; I dont know.)

System Usability Scale (SUS)

We asked you to use the assigned library and the following questions refer to the assigned library and its documentation. Please rate your agreement or disagreement with the following statements: (Strongly agree; agree; neutral; disagree; strongly disagree)

- I think that I would like to use this library frequently.
- I found the library unnecessarily complex.
- I thought the library was easy to use.

- I think that I would need the support of a technical person to be able to use this library.
- I found the various functions in this library were well integrated.
- I thought there was too much inconsistency in this library.
- I would imagine that most people would learn to use this library very quickly.
- I found the library very cumbersome to use.
- I felt very confident using the library.
- I needed to learn a lot of things before I could get going with this library.

Our usability scale

Please rate your agreement to the following questions on a scale from 'strongly agree' to 'strongly disagree.' (Strongly agree; agree; neutral; disagree; strongly disagree) Calculate the 0-100 score as follows: 2.5 * (5-Q_1 + $\sum_{i=2..10}(Q_i - 1)$); for the score, Q11 is omitted.

- I had to understand how most of the assigned library works in order to complete the tasks.
- It would be easy and require only small changes to change parameters or configuration later without breaking my code.
- After doing these tasks, I think I have a good understanding of the assigned library overall.
- I only had to read a little of the documentation for the assigned library to understand the concepts that I needed for these tasks.
- The names of classes and methods in the assigned library corresponded well to the functions they provided.
- It was straightforward and easy to implement the given tasks using the assigned library.
- When I accessed the assigned library documentation, it was easy to find useful help.
- In the documentation, I found helpful explanations.
- In the documentation, I found helpful code examples.

Please rate your agreement to the following questions on a scale from 'strongly agree' to 'strongly disagree.' (Strongly agree; agree; neutral; disagree; strongly disagree; does not apply)

- When I made a mistake, I got a meaningful error message/exception.
- Using the information from the error message/exception, it was easy to fix my mistake.

Demographics

- How long have you been programming in Python? (Less than 1 year; 1-2 years; 2-5 years; more than five years)
- How long have you been coding in general? (Less than 1 year; 1-2 years; 2-5 years; more than five years)
- How did you learn to code? [all that apply] (self-taught, online class, college, on-the-job training, coding camp)
- Is programming your primary job? If yes: Is writing Python code (part of) your primary job?
- Do you have an IT-security background? If yes, please specify.
- Please tell us your highest degree of education. (dropdown)
- Please tell us your gender. (female, male, other (please specify), decline to say)
- How old are you? (free text, check that the answer is a number)
- What country/countries do you live in / which country/-countries are you a citizen of? (dropdown)
- What is your occupation? (free text)

B. Security Errors

Table XIV details the different types of security errors made by our participants, across the libraries we tested and the tasks we assigned. Our definitions of security are discussed in Section III-H.

Symmetric Keygen	Key Size	Key in Plain	Weak Cipher	Weak Mode	Static IV	No KDF	Custom KDF	KDF Salt	KDF Algo.	KDF Iter.
PyCrypto	6	4	11	14	3	15	11	1	1	2
M2Crypto	2	2	0	0	7	4	2	2	1	1
cryptography.io	1	7	0	0	0	1	3	10	0	0
Keyczar	0	3	0	0	0	1	0	0	0	0
PyNaCl	0	2	0	0	0	1	17	1	1	0

Symmetric Encryption	No Enc.	Weak Algo.	Weak Mode	Static IV
PyCrypto	0	17	23	29
M2Crypto	0	0	1	9
cryptography.io	0	0	0	0
Keyczar	0	0	0	0
PyNaCl	0	0	0	0

Asymmetric Keygen	Key Size	Key in Plain	Weak Cipher	Weak Mode	Static IV	No KDF	Custom KDF	KDF Salt	KDF Algo.	KDF Iter.
PyCrypto	6	0	0	0	0	0	0	0	0	0
M2Crypto	6	0	0	0	0	0	0	0	0	0
cryptography.io	0	0	0	0	0	0	0	0	0	0
Keyczar	0	1	0	0	0	0	0	0	0	0
PyNaCl	0	3	0	0	0	0	7	0	0	0

Asymmetric Encryption	Key Size	Padding
PyCrypto	9	0
M2Crypto	6	1
cryptography.io	0	0
Keyczar	0	0
PyNaCl	0	0

Certificate Validation	Sig. Check	CA Flag Check	Hostname Check	Date Check
PyCrypto	1	1	1	1
M2Crypto	2	13	11	14
cryptography.io	4	7	7	7
Keyczar	0	0	0	0
PyNaCl	1	1	1	1

TABLE XIV
Security errors made by our participants, as categorized by our codebook.