Are Misuse-Resistant Cryptography Libraries Usable and Secure?

Noemi Glaeser*
nglaeser@umd.edu
University of Maryland
College Park, Maryland

Doruk Gür*
dgur1@umd.edu
University of Maryland
College Park, Maryland

Michael Pearson*
mpearso7@umd.edu
University of Maryland
College Park, Maryland

Abstract

We present a pilot study of miscreant, a nonce reuse-resistant Python cryptography library. In a between-subjects study, we analyze the functionality and security of code written by participants and measure the usability of miscreant. We compare the results to code written with PyCrypto, a popular Python cryptography library. With our results, we gain greater insight into misuse-resistant cryptography libraries and direction for a full-sized study and future work in usable cryptography libraries.

1 Introduction

Cryptography is widely used in applications all over the world. As TLS becomes the default for a lot of web traffic, especially in phone apps, developers need to perform cryptographic tasks such as encryption and certificate validation [1]. To develop an application that uses encryption, they often use cryptography libraries that implement the underlying cryptographic algorithms and provide (ideally easy-to-use) cryptographic functions. However, prior research found that cryptography libraries are often challenging to use, serving as a roadblock to code that is both functional and secure [7]. One reason that developers write insecure code with cryptography libraries is a lack of understanding of their inner workings. This can lead to security flaws such as nonce reuse or using nonrandom nonces [24].

We seek to understand if so-called "misuse-resistant" cryptography libraries are effective tools for implementing cryptography. Specifically, we want to answer the following questions: (1) Do they actually lead to more secure implementations than popular cryptography libraries? (2) Is usability sacrificed in making the libraries misuse-resistant?

We believe these are fundamental questions that need to be answered to promote secure code, because most insecurities in cryptographic code arise from the way in which a particular library is used. It is important to understand how to encourage secure implementations of cryptographic protocols when using external libraries. Misuse-resistant cryptography libraries are beginning to emerge in response to the pervasiveness of insecure cryptography implemented with popular libraries, but to our knowledge, there has been no empirical evaluation of software using these libraries nor of their usability.

*All authors contributed equally to this work.

To answer these questions we conducted a between-subjects online study comparing a misuse-resistant cryptography library, miscreant[1], with a popular cryptography library, PyCrypto[2]. We evaluated code that participants wrote with each library for functionality and security. In addition to evaluating the code, we measured the usability of each library with the well-established System Usability Scale[11].

The rest of the paper is organized as follows: In Section 2 we review related works. We describe our methodology in Section 3 and results in Section 4. We discuss the implications of our results in Section 5 and conclude in Section 6.

2 Related Work

Despite the existence of provably secure cryptographic algorithms, cryptography failures still occur. Egele et al. conducted a survey of over 11,000 apps in the Android App store [14] and found that 88% of them made at least one cryptography mistake. Furthermore, Heninger et al. studied RSA and DSA public keys and were able to recover a large number of private keys due to shared public key factors [18]. These common factors were attributed to insufficient system randomness from the kernel. Springall et al. analyzed methods for easily restarting TLS sessions and their implications for forward secrecy [22]. They found that in the event of a compromise of private keys, forward secrecy would be broken in most cases for at least several days and up to a few months. They highlight the catastrophic effects that can occur from cryptography failures.

Peter Gutmann discussed the possibility of developers misusing algorithms due to a lack of low-level cryptographic understanding, citing common mistakes such as ECB mode and improper initialization vectors [17]. To combat that he suggests that we "Provide crypto functionality at the highest level possible in order to prevent users from injuring themselves and others through misuse of low-level crypto functions with properties they aren't aware of" [17]. The Miscreant cryptography library seeks to do just that [8]. It provides resistance against nonce reuse, a common cryptography pitfall.

Miscreant is one implementation of the idea of "misuse-resistant authenticated encryption", first introduced by Rogaway and Shrimpton [21]. Their definition of misuse-resistance

[1] https://github.com/miscreant/miscreant.py
[2] https://www.dlitz.net/software/pycrypto/

deals with *initialization vector (IV)* misuse. An insufficiently random or repeated IV can compromise an authenticated encryption (AE) scheme. Their proposed solution, the *Synthethic IV (SIV)*, aims to preserve authenticity and privacy through minimal information leak even in the case of a badly chosen IV. SIV variants of AEs are widely used (two of which, AES-SIV and AES-PMAC-SIV, are part of miscreant library).

SIV, however, does not solve the nonce security problem. Bellare and Keelveedhi [10] discuss the impossibility of achieving universal nonce security in any of the AE schemes with key-dependent data and propose a transformation that could be used. They devise an attack that can recover the key of an AE scheme if the nonce is repeated or non-random, but which is thwarted by the incorporation of a random oracle. However, the proof of security is nontrivial, so this transformation is not widely used in misuse-resistance.

Alternatively, specific modifications can be made on a per-scheme basis to incorporate misuse-resistance. Gueron and Lindell [16] modified the GCM scheme based on Rogaway and Shrimpton's work and created GCM-SIV, which is significantly faster than original SIV-based AE. Compared to other generic schemes in OpenSSL [23] GCM-SIV is slightly slower for encryption and comparable for decryption. It is faster than GCM for short encryptions. The scheme is in the process of being standardized, but is not included in our study as it is not present in a generic library for a non-expert to test it.

Forler et al. [15] incorporated this paradigm into an existing API by creating AE and secure nonce generation functions based on Ada's cryptography library libadacrypt [6]. Their solution combines counter values and randomness for nonce generation, giving resistance against nonce reuse, but do not evaluate their tool's usability. Although their solution is less error prone than other measures, the authors themselves admit that design is counter-intuitive.

Thus, an opposite but complementary approach is to create APIs that are easier to understand and use. These will ideally guide end users, who are not cryptography experts, around common pitfalls. In 2014, Das et al. [13] surveyed six cryptographic libraries across five programming languages, describing seven common issues[3], how they are handled in each library, and a best practice in each case. One of these libraries is PyCrypto, which we use as the control in our study. Unlike Das et al., we conduct a user study to assess real programmers' reactions to our selected libraries and evaluate the security of real solutions implemented using them.

Nadi et al. [20] take a step closer to the developers by assessing the usability of the Java Cryptography API through the analysis of StackOverflow posts, developer surveys, and inspection of code on GitHub. Though our study also interacts with developers directly, unlike Nadi et al., we also ask our participants to complete predefined tasks using the libraries, which can then be compared for security across developers and libraries. We also focus on Python instead of Java, since Python is a more popular programming language today (ranked 1^{st} and 3^{rd} in the PYPL[3] and TIOBE[2] indices, respectively).

The design of our study mirrors Acar et al.'s 2017 study [7] on the usability of cryptographic APIs, but differs in the APIs we evaluate: we ask our participants to use a misuse-resistant library (miscreant), which is based on the paradigm of misuse-resistant cryptography discussed above. We keep most of the other elements of the study (tasks, SUS scale for usability, etc.) the same to enable comparison of miscreant to the other libraries discussed by Acar et al. and not used in this study due to time and size limitations. This methodology has been used in other studies as well, including Mindermann et al.'s 2018 study of Rust cryptography APIs [19], which gives us confidence in the design.

All previous works drew similar conclusions: API simplicity is not enough for security; well-chosen examples can be just as, if not more, effective than comprehensive documentation; and developers hardly consider the security of their implementations, but prioritize functionality. Our study aims to understand how these principles could apply to the few existing misuse-resistant libraries in the hopes of providing some paradigm-specific guidance to augment their usability and documentation as they develop.

3 Methods

3.1 Recruitment

We recruited participants with Python experience from the University of Maryland (UMD) by posting an advertisement on a Piazza page accessible to a large number of undergraduate students. To ensure that participants had adequate Python skills for the study, they first completed a prescreen with a small Python task. Participants were not filtered based on their prior cryptography knowledge.

3.2 Study Design

We designed an IRB-approved study that consisted of a prescreen and main study. Participants who successfully completed the prescreen were invited to participate in the main study, which included two programming tasks and an exit survey. The study poses limited ethical concerns, but our methodology addresses privacy concerns by hosting the prescreen and main study on Qualtrics.

3.2.1 Prescreen. Interested participants received a screening test consisting of both a background survey and a programming task. The background survey asked users for their prior experience with Python (last time it was used, the nature of the projects previously done) and programming in

[3]IV reuse, method defaults, bloat caused by outdated/insecure methods, omission of newest features, documentation, sample code, and primary programming language.

general. Participants were then given 20 minutes to complete a simple programming task designed to assess general familiarity with Python and were allowed to access any documents they needed as they worked. Presenting a functional solution that showed previous experience was enough to qualify for the main study. The survey questions and prescreen programming task can be found in Appendices A and D, respectively.

3.2.2 Programming Tasks. After qualifying, participants were randomly separated into two groups (regular and misuse-resistant) and were presented with two tasks (secure key generation/storage and encryption) similar to the ones described by Acar et al. [7]. Each task consisted of skeleton Python code describing the functionality requirements and the cryptography library to be used. Participants in the misuse-resistant group were asked to use miscreant whereas participants in the "regular" group used PyCrypto. They were given full documentation access to solve any possible problems relating to the libraries. Participants' solutions to the given tasks were stored and later analyzed for both security and functionality. The programming tasks can be found in Appendix D.

3.2.3 Exit Survey. After the programming section, participants were asked to fill out an exit survey so we could gather data on their experience using the libraries in question. For each task, a set of questions prompted users for their documentation usage, and their perceived security and perceived functionality of their solutions on a 5-point Likert scale followed by a set resembling Acar et al.'s [7] new usability scale. After both tasks were complete, participants were asked to rate the usability of their assigned library using the well-established System Usability Scale (SUS)[11] and were asked about their security background. The exact questions can be found in Appendix A.

4 Results

4.1 Participants

We recruited seven participants to complete the prescreen from our advertisement. Of the initial seven participants, six completed the prescreen. We invited five of those participants to participate in the study, which four completed. (The last participant started the assigned task but did not finish at the end of data collection period.) Of the four participants who completed the study, two were assigned PyCrypto (P1 and P2) and two were assigned miscreant (M1 and M2). Participants who completed the study received a $20 Amazon gift card for their time.

4.2 Functionality, Security, and Usability Scores

The security choices to be made when using each library are shown in Table 1. Implementations were scored binarily on functionality and security. Solutions received a "1" for functionality according to the standard in [7]: if they ran "without errors, passed the tests, and completed the assigned task". Unlike [7] and because of the paucity of data, we evaluated the security of both functional and non-functional implementations. Security was broken into several parameters, each scored "1" (if a secure choice was made) or "0" (otherwise). We assigned "-" when the parameter was not applicable. For example, P2 did not use a key derivation function (KDF), so we could not score the KDF's salt, pseudorandom function (PRF), or iterations. Solutions are considered secure iff every parameter received a score of "1". By this standard, none of the solutions for Task 1 (key generation and secure storage) are secure and only two of the four solutions for Task 2 (encryption), namely P2's and M1's, are secure.

Three researchers independently scored each task and assigned functionality and security scores for each security parameter. They agreed with a Krippendorff's Alpha of 0.841, then met to resolve disagreements [4].

Neither library performed particularly well. With PyCrypto, half of the solutions to the key generation and storage task were functional and none were secure: P1 did well but was foiled when using the KDF, since they used a static salt and PyCrypto's insecure defaults for the PRF choice and iterations. P2 faced issues in the same area, using SHA-256 instead of a real KDF to obtain a hash of the password. Additionally, they confused the randomly generated symmetric key with the password-derived hash and stored the hash instead of the key. In the encryption and decryption task, half the solutions were functional again, but all were secure. Just as in the first task, P1 used incorrect import statements which made the code fail to run (we discuss this in more detail in Section 5.1). The median SUS score for PyCrypto was 46.25. Since SUS scores below 68 are considered below average, this is undesirable [5].

Miscreant had similar results for the key generation and storage task. Half of the solutions were functional and half of the solutions were secure. M1 was very close to a fully secure solution, but used a static salt for the KDF. M2 grew confused on several fronts and seemed to miss the purpose of miscreant altogether; we discuss this further in Section 5.2. In the encryption and decryption task, both were functional and half were secure, since M2 did not use a nonce. Note, however, that this is exactly the mistake miscreant was created to mitigate, and in theory this solution is still secure. The median SUS score for miscreant was 50, which is once again below average. Details about which choices were regarded as secure or insecure can be found in Appendix C, with breakdowns of the assigned codes for each participant in Appendix D.

4.3 Regression Models

We also tried to build regression models based on our restricted data. Our goal was to have a general idea on the possible implications if we had sufficient data to process. Hence, regression analysis was done as a demonstration

Table 1. Security choices required by our chosen libraries (based on Table III of [7]). ● means the user must make a secure choice and ○ that no choice is required to obtain a secure solution.

	KeyGen		Key Storage					Key Derivation				Encryption		
	size	random[+]	plain/enc	size[+]	algo	mode	IV	use[+]	salt	PRF	iterations	algo	mode	IV
PyCrypto[†]	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Miscreant	○	○	●	○	○	○[*]	●[**]	●[***]	●[***]	●[***]	●[***]	○	○	●

[†] Taken from Table III of [7].

[+] Columns not included in Table III of [7].

[*] Miscreant implements AES-SIV, which is not a mode per se precludes use of other (potentially insecure) modes.

[**] Users must choose a nonce, but encryption is resistant to nonce misuse so even a poor choice should not impact security.

[***] Library does not include key derivation function, so users are expected to use hashlib.

of our approach and the resulting models should not be considered meaningful. We used logistic regression for variables with binary outcomes such as functionality and linear regression for the rest. Self-assessed functionality and self-assessed security were considered as confidence scales and thus treated as non-binary outcomes for regression analysis. Final models for each category were chosen based on the Akaike Information Criterion[12].
A majority of the models did not produce any statistically significant results (defined here as factors with p-values lower than 0.05). This was expected considering none of the logistic regression models converged and both of them resulted in models with high p-value factors. While not being statistically significant or meaningful, a summary of final models can be found in Appendix E.

5 Discussion

5.1 Limitations

Our study has several limitations. First, our sample size was very small and not representative of the population that we are studying since we only studied students at UMD. Meaningful conclusions cannot be made about the functionality, security, usability scores, and regression analyses run on the data.

Second, we only compared two specific tasks using one popular library with one misuse-resistant library. While our tasks were concerned with symmetric encryption, cryptographic algorithms include a a variety of additional operations, and hence our results do not generalize to cryptography libraries in general.

Third, we did not make it clear to participants that they could use additional libraries or otherwise modify the provided import statement for the library they were asked to use. P1's code wasn't functional because the participant did not realize the import statement could be modifed. They tried to use legitimate functions from PyCrypto, but the code did not run because it couldn't properly find the functions. M1 noted in the response that they weren't sure if they were allowed to use hashlib for their KDF since miscreant did not provide one.

Fourth, participants had to install the libraries on their own machines, which could have affected their perception of how usable the library was.

Finally, a social desirability bias could affect the usability scores as participants might report the libraries easier to use than they actually are, or otherwise over- or under-report certain phenomena.

5.2 Takeaways

Miscreant is not the solution. In terms of both usability and security, miscreant did not present an advantage over PyCrypto based on participants' solutions. This may be expected considering miscreant is not a fully featured library with multiple functionalities but a specialized encryption tool with only three functions: key generation, encryption, and decryption. This made the first task rather difficult to complete and even required turning to outside libraries. It is also worth noting that while PyCrypto is continuously managed, the developers of miscreant do not seem to be actively working on the library. In fact, the ad hoc feel of miscreant (whose entire documentation is hosted in a short section of a GitHub readme file) may have contributed to the confusion of at least one participant: failing to implement some of the requirements of the first task, M2 imported PyCrypto and finished the task in that library instead. M2 was lead to PyCrypto by an online tutorial for encrypting files in Python. Lack of features expected from a standard cryptography library combined with drastic differences in documentation in terms of style, size, and typing keeps miscreant from being the next big solution to cryptography misuse.

Misuse-resistant libraries do not address the root problem. While miscreant is not the solution to misuse in general, the concept of misuse-resistant libraries seems to overlook the more immediate problem in insecure decisions made by users. The main feature of miscreant is to provide misuse resistance to the written code, and yet it didn't prevent M2

from not including a nonce to the solution. A majority of users lack the proper background to fully understand requirements and capabilities of the API at hand, which would make misuse-resistant features of new libraries useless. To prevent cryptography misuse, APIs that discourage users from the inclusion of insecure options should be favored.

Catering to backwards compatibility leads to the inclusion of insecure choices and defaults. Due to legacy reasons and compatibility issues, PyCrypto and many other libraries still contain insecure algorithms, modes, and default values for parameters. Participant P1 failed to present a secure KDF as part of the key generation due to use of an insecure algorithm with an insecure default salt and insecure number of iterations included within PyCrypto. As long as these deprecated measures remain available in cryptographic APIs, it will be possible to unintentionally use these tools to produce weak solutions.

5.3 Future Work

Improve cryptography APIs. To continue improving usability of cryptography APIs for more secure solutions, the APIs themselves have to be changed to reduce user mistakes. Our work showed the source of misuse often comes from the perception and background of users. Therefore, to prevent future mistakes, removing choice when users don't have the necessary information to make a *secure* one would improve APIs and the security of solutions written with them.

Sanitizing defaults. Removing unnecessary choices from users only will not suffice as the remaining choices also have to be sanitized to accommodate more secure algorithms and defaults. Many cryptographic libraries still include deprecated algorithms (e.g. DES) or insecure default parameters (like non-random salts) which leaves the user with a chance of using these insecure approaches. APIs should remove these options to accommodate new, useful paradigms like misuse resistance. A tool like miscreant could be more useful if incorporated into a mainstream library like PyCrypto.

Improving documentation. Good APIs themselves will not be enough to provide fully secure solutions. To help users, the complementary material should also be updated in terms of clarity and usability. Users should be assumed to have no prior knowledge on the subject matter; details and possible mistakes should be outlined in detail. API documentation should also point users to proper places for features that are not included within the API in order to prevent errors caused by a lack of features considered "out of scope" by the API. This could have prevented KDF usage errors in miscreant.

6 Conclusion

Cryptography is commonly used in a variety of applications, bringing with it the potential for a myriad of errors and security weaknesses. Misuse-resistant libraries are one of the proposed solutions for reducing the number of developer-introduced errors. In this pilot study, we set out to find whether this new generation of libraries is more usable and yields more secure solutions than the traditional counterparts by comparing two libraries: miscreant and PyCrypto. Since this was only a pilot study, a full-scale study needs to be done to obtain more meaningful data. However, the results of our pilot study suggest misuse-resistant libraries might not present an advantage in terms of usability and security, as the main factors affecting secure, functional solutions lie with the users. For this reason, we instead recommend improving existing libraries both in terms of features and supplementary material to prevent misuse-based problems.

Acknowledgments

We would like to thank Michelle Mazurek and Omer Akgul for their guidance during the semester and plentiful and helpful feedback throughout the course of this project. We are also grateful to our anonymous classmates for their insightful comments and questions during the process. Finally, we thank Ming Lin and the Computer Science Department at UMD for providing the funding necessary to complete this research.

References

[1] [n.d.]. 80% of all Android apps are encrypting traffic by default. https://www.zdnet.com/article/80-of-all-android-apps-are-encrypting-traffic-by-default/

[2] [n.d.]. Latest news. https://www.tiobe.com/tiobe-index/

[3] [n.d.]. PYPL PopularitY of Programming Language index. http://pypl.github.io/PYPL.html

[4] [n.d.]. ReCal for Ordinal, Interval, and Ratio Data (OIR). http://dfreelon.org/utils/recalfront/recal-oir/

[5] [n.d.]. System Usability Scale (SUS). ([n. d.]). https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html

[6] 2011. Ada-Crypto-Library. (2011). https://github.com/cforler/Ada-Crypto-Library.

[7] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2017. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 154–171.

[8] Tony Arcieri. 2017. Introducing Miscreant A misuse-resistant encryption library, available for the following languages:. (2017). miscreant.io.

[9] Elaine Barker and Allen Roginsky. 2019. NIST Special Publication 800-131A Revision 2: Transitioning the Use of Cryptographic Algorithms and Key Lengths. https://csrc.nist.gov/publications/detail/sp/800-131a/rev-2/final

[10] Mihir Bellare and Sriram Keelveedhi. 2011. Authenticated and misuse-resistant encryption of key-dependent data. In *Annual Cryptology Conference*. Springer, 610–629.

[11] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[12] Kenneth P Burnham and David R Anderson. 2004. Multimodel inference: understanding AIC and BIC in model selection. *Sociological methods & research* 33, 2 (2004), 261–304.

[13] Somak Das, Vineet Gopal, Kevin King, and Amruth Venkatraman. [n.d.]. Cryptographic Misuse of Libraries. ([n. d.]). https://pdfs.semanticscholar.org/03ca/653e5ac26bd5575f7ef578aa3c2c7b964313.pdf

[14] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *20th ACM Conference on Computer and Communications Security*. ACM, 73–84.
[15] Christian Forler, Stefan Lucks, and Jakob Wenzel. 2012. Designing the API for a cryptographic library: A misuse-resistant application programming interface, Vol. 7308. 75–88. https://doi.org/10.1007/978-3-642-30598-6_6
[16] Shay Gueron and Yehuda Lindell. 2015. GCM-SIV: full nonce misuse-resistant authenticated encryption at under one cycle per byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 109–119.
[17] Peter Gutmann. 2002. Lessons Learned in Implementing and Deploying Crypto Software. In *11th USENIX Security Symposium*. 315–325.
[18] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *21st USENIX Security Symposium*. IEEE.
[19] Kai Mindermann, Philipp Keck, and Stefan Wagner. 2018. How Usable are Rust Cryptography APIs? *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (July 2018), 143–154. https://doi.org/10.1109/QRS.2018.00028 arXiv: 1806.04929.
[20] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: why do Java developers struggle with cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, Austin, Texas, 935–946. https://doi.org/10.1145/2884781.2884790
[21] Phillip Rogaway and Thomas Shrimpton. 2006. A provable-security treatment of the key-wrap problem. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 373–390.
[22] Drew Springall, Zakir Durumeric, and J. Alex Halderman. 2016. Measuring the Security Harm of TLS Crypto Shortcuts. In *2016 Internet Measurement Conference*. 33–47.
[23] The OpenSSL Project. 2003. OpenSSL: The Open Source toolkit for SSL/TLS. (2003). www.openssl.org.
[24] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *29th Usenix Security Symposium*.

A Survey Text

A.1 Prescreen

- How long have you been programming? (<1 year; 1-2 years; 2-5 years; >5 years)
- When was the last time you programmed in Python? (In the last week; 1 week to <1 month ago; 1 month to <6 months ago; 6 months to <1 year ago; ≥ 1 years ago)
- To what extent do you use Python? (Check all that apply) (I have used Python for class; I have used Python outside of class for personal projects; I have used Python on a project or application for payment; I have used Python in a project or application published on GitHub or a similar platform and available for public use)

A.2 Exit Survey

The following shows the questions used in the exit survey. Besides the first two task-based ones, each of the questions asked participants for their agreement with the given statement on a 5-point Likert scale. Attention checks are omitted.

Task-Based Questions. Briefly describe what you have been asked to do in this task.
What kind of documentation (if any) did you use to accomplish the first task? (Official API Documentation; StackOverflow or other similar platforms; Google or other similar search engines; Other (Please explain); I didn't use any kind of documentation.)
Now think about your development process and solution. Please rate your agreement/disagreement for the following accordingly.

- I think my solution satisfies the required functionality.
- I believe my solution is secure and can probably withstand the majority of the most common attacks.
- I had to go through and use a significant portion of the given API to accomplish this task.
- I believe I can reuse my written code with minor adjustments if I have to accomplish a similar task.
- Security and development concepts required for this task were previously known to me.
- I could easily grasp required security and development concepts after partially reading the documentation I used for the given API.
- The documentation I used had useful examples and explanations for functionality.
- The naming of classes and functions in the API were intuitive and straightforward.
- If I encountered an exception or an error, the descriptive text was easy to understand.
- It was easy to fix my errors based on encountered error or exception messages.
- It was easy to accomplish the assigned task with the given API.

System Usability Scale (SUS). Now based on your previous answers, please rate your agreement/disagreement on following statements about the general usability of the assigned library.

- I think that I would like to use this API frequently.
- I found the API unnecessarily complex.
- I thought the API was easy to use.
- I think that I would need the support of a technical person to be able to use this API.
- I found the various functions in this API were well integrated.
- I thought there was too much inconsistency in this API.
- I would imagine that most people would learn to use this system very quickly.
- I found the API very cumbersome to use.
- I felt very confident using the API.

- I needed to learn a lot of things before I could get going with this API.

Security Background.

- Do you have a background in security, cryptography or a similar subject? (Yes; No)
- Please tell us how long you were working on that subject. (This question only appears if the previous question is answered "Yes")(<1 year; 1-2 years; 2-5 years; >5 years)

B Programming Tasks

B.1 Prescreen

You will have 20 minutes to complete the following task. You are free to use any resources (books, search engine, documentation, etc.) to help you.

Write a program that populates a list of user-provided size (at most 100) with random numbers between 1 and 100 (inclusive). Calculate the average over the elements in the list and write this number to a file named "averages.txt". If a user runs the program again, append the new average to the same file.

Running the program for a random list of size 10 would look something like this: python averages.py 10

After three runs of the program, "averages.txt" might look something like this:
47.6
32.5
55.1

B.2 Task 1 (Key Generation and Storage)

PyCrypto Text. Goal: Use the PyCrypto library to create a secure symmetric key and safely store it in a file named "my_key" that is protected with the password "my_password". You may write any helper functions that you need.
Documentation: https://www.dlitz.net/software/pycrypto/api/current/

Please download the skeleton Python file and fill it in.

PyCrypto Skeleton Code. The given skeleton code is as follows:

```
# Key Generation and Storage
# Goal: Use PyCrypto to create a secure symmetric key and safely store it in a file named "my_key" that is
# protected with the password "my_password". You may write any helper functions that you need.

# Documentation: https://www.dlitz.net/software/pycrypto/api/current/

import Crypto

def keygen():
    '''
    Purpose:
    Create a symmetric key and store it in a file.

    Arguments:
    N/A

    Return value:
    N/A

    Notes:
        - If you used additional resources besides the official documentation
          (e.g. search engine, StackOverflow), please paste the links here:
    '''

    # This is where your code goes

# This will test the code for this task
keygen()
print("Task completed! Please continue.")
```

Miscreant Text. Goal: Use the Miscreant library to create a secure symmetric key and safely store it in a file named "my_key" that is protected with the password "my_password". You may write any helper functions that you need.
Documentation: https://github.com/miscreant/miscreant.py#api

Please download the skeleton Python file here and fill it in:

Miscreant Skeleton Code. The given skeleton code is as follows:

```
# Key Generation and Storage
# Goal: Use miscreant to create a secure symmetric key and safely store it in a file named "my_key"
# that is protected with the password "my_password". You may write any helper functions that you need.

# Documentation: https://github.com/miscreant/miscreant.py#api

import miscreant

def keygen():
    '''
    Purpose:
    Create a symmetric key and store it in a file.
```

```
13
14      Arguments:
15      N/A
16
17      Return value:
18      N/A
19
20      Notes:
21          - If you used additional resources besides the official documentation (e.g. search engine, StackOverflow), please paste the links here:
22      '''
23
24      # This is where your code goes
25
26  # This will test the code for this task
27  keygen()
28  print("Task completed! Please continue.")
```

B.3 Task 2 (Encryption)

PyCrypto Text. Goal: Use the PyCrypto library to encrypt and then decrypt the files "message.txt", "message1.txt", and "message2.txt" using the symmetric key from the previous task. You may write any helper functions that you need. Documentation: https://www.dlitz.net/software/pycrypto/api/current/

Please download the skeleton Python file and fill it in:

PyCrypto Skeleton Code. The given skeleton code is as follows:

```
1   # Encryption and Decryption
2   # Goal: Use PyCrypto to encrypt and then decrypt the files "message.txt", "message1.txt", and "message2.txt"
3   # using the symmetric key from the previous task. You may write any helper functions that you need.
4
5   # Documentation: https://www.dlitz.net/software/pycrypto/api/current/
6
7   import Crypto
8
9   def encrypt(filename):
10      '''
11      Purpose:
12          Use your symmetric key to encrypt a file. Output the encrypted text to a new file called "[filename]_enc.txt".
13
14      Arguments:
15          filename: The name of the file to encrypt.
16
17      Return value:
18      N/A
19
20      Notes:
21          - The files to encrypt can be found at ./message.txt, ./message1.txt, and ./message2.txt
22          - If you used additional resources besides the official documentation (e.g. search engine, StackOverflow), please paste the links here:
23      '''
24
25      # This is where your code goes
26
27  def decrypt(filename):
28      '''
29      Purpose:
30          Use your symmetric key to decrypt a file. Output the encrypted text to a new file called "[filename]_dec.txt".
31
32      Arguments:
33          filename: The name of the file to decrypt.
34
35      Return value:
36          N/A
37
38      Notes:
39      - Decrypt the files you encrypted, which should be found at ./message_enc.txt, ./message1_enc.txt, and ./message2_enc.txt
40      - If you used additional resources besides the official API (e.g. search engine, StackOverflow), please paste the links here:
41      '''
42
43      # This is where your code goes
44
45
46
47  # This will test the code for this task
48  def compare_files(file1, file2):
49      with open(file1, "r") as f1:
50          with open(file2, "r") as f2:
51              return f1.read() == f2.read()
52
53  encrypt("message.txt")
54  decrypt("message_enc.txt")
55  assert compare_files("message.txt", "message_dec.txt"), "message did not decrypt correctly"
56
57  encrypt("message1.txt")
58  decrypt("message1_enc.txt")
59  assert compare_files("message1.txt", "message1_dec.txt"), "message1 did not decrypt correctly"
60
61  encrypt("message2.txt")
62  decrypt("message2_enc.txt")
```

```
assert compare_files("message2.txt", "message2_dec.txt"), "message2 did not decrypt correctly"

print("Task completed! Please continue.")
```

Miscreant Text. Goal: Use the Miscreant library to encrypt and then decrypt the files "message.txt", "message1.txt", and "message2.txt" using the symmetric key from the previous task. You may write any helper functions that you need. Documentation: https://github.com/miscreant/miscreant.py#api

Please download the skeleton Python file here and fill it in:

Miscreant Skeleton Code. The given skeleton code is as follows:

```
# Encryption and Decryption
# Goal: Use miscreant to encrypt and then decrypt the files "message.txt", "message1.txt", and "message2.txt"
# using the symmetric key from the previous task. You may write any helper functions that you need.

# Documentation: https://github.com/miscreant/miscreant.py#api

import miscreant

def encrypt(filename):
    '''
    Purpose:
        Use your symmetric key to encrypt a file. Output the encrypted text to a new file called "[filename]_enc.txt".

    Arguments:
        filename: The name of the file to encrypt.

    Return value:
    N/A

    Notes:
        - The files to encrypt can be found at ./message.txt, ./message1.txt, and ./message2.txt
        - If you used additional resources besides the official documentation (e.g. search engine, StackOverflow), please paste the links here:
    '''

    # This is where your code goes

def decrypt(filename):
    '''
    Purpose:
        Use your symmetric key to decrypt a file. Output the encrypted text to a new file called "[filename]_dec.txt".

    Arguments:
        filename: The name of the file to decrypt.

    Return value:
        N/A

    Notes:
    - Decrypt the files you encrypted, which should be found at ./message_enc.txt, ./message1_enc.txt, and ./message2_enc.txt
    - If you used additional resources besides the official API (e.g. search engine, StackOverflow), please paste the links here:
    '''

    # This is where your code goes



# This will test the code for this task
def compare_files(file1, file2):
    with open(file1, "r") as f1:
        with open(file2, "r") as f2:
            return f1.read() == f2.read()

encrypt("message.txt")
decrypt("message_enc.txt")
assert compare_files("message.txt", "message_dec.txt"), "message did not decrypt correctly"

encrypt("message1.txt")
decrypt("message1_enc.txt")
assert compare_files("message1.txt", "message1_dec.txt"), "message1 did not decrypt correctly"

encrypt("message2.txt")
decrypt("message2_enc.txt")
assert compare_files("message2.txt", "message2_dec.txt"), "message2 did not decrypt correctly"

print("Task completed! Please continue.")
```

C Codebook for Implementations

Unless otherwise marked, these choices are taken from [7].

KeyGen:

- Size: Generated key is a secure size (≥ 128 bits [9])
- Random: Key is generated using a good source of randomness (`os.urandom()` or other secure sources of randomness)

Key Storage:

- Plain/Enc: Key is stored encrypted
- Size: Key is encrypted with a key of a good size (same as above)
- Algo: A secure algorithm is used to encrypt the key (AES)
- Mode: A secure mode of AES is used to encrypt the key (CBC, CTR, CFB)
- IV: A secure IV is used to encrypt the key (not static, empty, or zero)

Key Derivation:

- Used: A secure KDF (e.g. PBKDF2) is used to derive a strong key from the password
- Salt: A secure salt is used for the KDF (not static, empty, or zero)
- PRF: A secure PRF is used for the KDF (nothing less than HMAC-SHA1)
- Iterations: Enough iterations of the KDF are performed (at least 10,000)

Encryption:

- Algo: A secure algorithm is used to encrypt the key (AES)
- Mode: A secure mode of AES is used (CBC, CTR, CFB)
- IV: A secure IV is used (not static, empty, or zero)

D Functionality and Security Codes

Tables 2 and 3 summarize the functionality and security of the implementations.

E Regression Results

Tables 4, 5, 6, 7, and 8 summarize final models, detailing the resulting factors with their odds ratios or coefficients, confidence intervals, and p-values. Higher odds ratios and higher coefficient absolute values indicate higher effect on the model and p-values lower than 0.05 show statistical significance. While these results are not meaningful or statistically significant due to sample size, it can be seen that the resulting models favor factors other than library choice.

Factor	O.R.	C.I.	p-value
Task: Key storage	0.400	[-4.028, 2.194]	0.564
Library: PyCrypto	0.856	[-3.820, 3.509]	0.934
Security background	1.833	[-3.956, 5.169]	0.795

Table 4. Final model for functionality of the solutions.

Factor	O.R.	C.I.	p-value
Task: Key storage	0.730	[-3.538, 2.907]	0.848
Library: PyCrypto	1.160	[-3.277, 3.574]	0.932
Document: Other	1.247	[-6.240, 6.682]	0.947
Security background	1.052	[-5.091, 5.192]	0.985

Table 5. Final model for security of the solutions.

Factor	Coeff.	C.I.	p-value
Task: Key Storage	-2.143	[-5.154, 0.868]	0.092
Library: PyCrypto	-1.57	[-6.930, 3.787]	0.334
Document: StackOverflow	-2.429	[-9.491, 4.633]	0.277
Document: Search Engine	3.429	[-0.830, 7.687]	0.074
Document: Other	-1.143	[-6.641, 4.355]	0.465
Security Background	2.571	[-1.687, 6.830]	0.122

Table 6. Final model for participants' self-assessed functionality of their results with $R^2 = 0.929$.

Factor	Coeff.	C.I.	p-value
Task: Key storage	-0.750	[-1.901, 0.401]	0.145
Library: PyCrypto	-1.125	[-3.034, 0.784]	0.177
Document: StackOverflow	-2.125	[-4.762, 0.512]	0.089
Security background	3.500	[1.872, 5.128]	0.004*

Table 7. Final model for participants' self-assessed functionality of their results with $R^2 = 0.913$. The * denotes statistically significant result.

Factor	Coeff.	C.I.	p-value
Library: PyCrypto	5.833	[-5.677, 17.344]	0.232
Document: StackOverflow	-11.667	[-34.688, 11.354]	0.232
Document: Google	15.000	[-14.449, 44.449]	0.230
Security background	35.000	[9.497, 60.503]	0.019*

Table 8. Final model for participants' self-assessed functionality of their results with $R^2 = 0.808$. The * denotes statistically significant result.

Participant	Functionality	KeyGen		Key Storage					Key Derivation			
		Size	Random	Plain/Enc	Size	Algo	Mode	IV	Used	Salt	PRF	Iterations
P1	0	1	1	1	1	1	1	1	1	0	0	0
P2	1	1	0	1	1	1	1	1	0	-	-	-
M1	1	1*	1*	1	1*	1*	1*	1	1	0	1	1
M2	0	1*	1*	0	-	-	-	-	0	-	-	-

* no choice was required to obtain a secure solution

Table 2. Functionality and security of participant solutions for Task 1 (key generation and storage).

Participant	Functionality	Encryption		
		Algo	Mode	IV
P1	0	1	1	1
P2	1	1	1	1
M1	1	1*	1*	1
M2	1	1*	1*	0

* no choice was required to obtain a secure solution

Table 3. Functionality and security of participant solutions for Task 2 (encryption).