"Do this! Do that!, And nothing will happen" Do specifications lead to securely stored passwords?

Joseph Hallett, Nikhil Patnaik, Benjamin Shreeve and Awais Rashid
University of Bristol

***Abstract*—Does the act of writing a specification (how the code should behave) for a piece of security sensitive code lead to developers producing more secure code? We asked 138 developers to write a snippet of code to store a password: Half of them were asked to write down a specification of how the code should behave before writing the program, the other half were asked to write the code but without being prompted to write a specification first. We find that explicitly prompting developers to write a specification has a small positive effect on the security of password storage approaches implemented. However, developers often fail to store passwords securely, despite claiming to be confident and knowledgeable in their approaches, and despite considering an appropriate range of threats. We find a need for developer-centered usable mechanisms for telling developers how to store passwords: lists of what they *must* do are not working.**

I. Introduction

Developers struggle to store passwords securely. Naiakshina et al. have repeatedly shown that developers do not build in security unless explicitly asked to do so (and even then typically do so poorly) [3], [4], [5]. In organizations, one can support developers coding securely through code review, and acceptance testing—but not all developers work in teams and many work alone on their own projects [6], [7].

Developers continue to seek guidance on how to handle passwords. In a survey of developers' posts on Stack Overflow (a popular developer question and answer site) Barua et al. found that posts related to authentication and security (including password storage) and were one of the top 20 topics on the site and accounted for 2.1% of all questions [8]. Furthermore in a survey of just security-focussed posts on Stack Overflow, Yang et al. found that the most viewed of all security-focussed posts related to passwords [9], with each post viewed on average 2,731 times. Whilst there are alternatives to passwords [10], many developers still appear to be working with passwords and implementing password storage in their apps and software. As well as working with passwords they are seeking guidance on how to do it *right*.

It is well established that writing a specification before implementing it leads to code that is of a higher quality [11], [12], [13], [14], [2], [15]. Since specification writing is beneficial for *quality*, does the act of writing one also improve developers' security practices? Naiakshina's work suggests that developers only consider security aspects if explicitly prompted [5]; but if we try to continuously prompt developers to *work securely* we risk security fatigue [16], [17]. Since specification is an established developer practice, this paper seeks to explore whether the act of writing any form of specification primes developers to program securely: in other words whether giving developers time to make a plan (however formally or informally) leads them to either recalling more about how to store passwords, or to recall that a standard exists and to check.

To test this we recruited 138 developers from an online platform for recruiting participants for studies, and asked them to write code to store a password in whatever language they were most comfortable with. Half the developers were asked to write down a specification—any form of specification from a formal definition [18] to a prose description [19]—of how they would implement this before they were allowed to write their solution, the rest were allowed to write their implementation immediately. We scored the security of their implementations using Naiakshina's end-user password storage criteria [3] (Figure 1), which is itself based on NIST SP 800-63-3 [20], and analyzed their written justifications of their choices and threats considered.

Specifically, we address the following research questions:

RQ1 Does specification writing lead to a measurable improvement in password storage method?

RQ2 What approaches do developers take when implementing password storage and what do they typically remember and forget?

RQ3 How do developers justify their implementation approach and what threats do they consider?

Our key findings are as follows:

- Developers who were explicitly prompted to write a specification, stored their passwords *slightly* more securely than those who were not prompted ($p = 0.027$, $r_{rb} = 0.209$).
- Only 38% of developers remembered to hash passwords, 14% remembered to salt them, but other secure password storage practice was largely absent (Figure 1).
- Developers think they are storing passwords correctly, but their scores according to Naiakshina's criteria (Figure 1) do not indicate best practice.
- If given time to reflect, some developers do realize that there are threats to stored passwords and that their solutions may not be secure.

Our novel insights are in examining whether specification is a useful tool for priming for security related tasks, and how developers justify the code they write and the threats they consider with respect to the specifications they write.

The quote in the title is attributed to Harry S. Truman [1], and is used as the opening quote to chapter 6 in *The Mythical Man-Month* [2].

- The end-user password is salted (+1) and hashed (+1).
- The derived length of the hash is at least 160 bits long (+1).
- The iteration count for key stretching is at least 1,000 (+0.5) or 10,000(+1) for PBKDF2 and at least 2^{10} for bcrypt (+1).
- A memory-hard hashing function is used (+1).
- The salt value is generated randomly (+1).
- The salt is at least 32 bits in length (+1).

Fig. 1. Naiakshina's end-user password storage assessment criteria [3], copied verbatim. A score $\geq \mathbf{6}$ indicates industrial best practice.

Analyzing the developer's rationale suggests that whilst some developers consider appropriate threats, their knowledge of best practice is out-of-date and that current cryptographic guidelines [20]. Providing lists of what developers must do is not working. Instead we must fit the task to the developer and provide usable mechanisms for password storage.

II. Background and Related Work

A. Benefits of specification

The benefits of program specification are well established in both the academic and engineering communities. Spolsky notes their benefit saying:

> "If you don't have a spec, you will always spend more time and create lower quality code." [13]

Brooks Jr. also notes the benefit of a specification:

> "Careful function definition, careful specification, and the disciplined exorcism of frills of function and flights of technique all reduce the number of system bugs that have to be found." [2]

Dromey suggested that quality models and requirements specifications could lead to an improvement in software quality [21]. Haigh and Landwehr have suggested that by building code to security specifications (drawing analogy to US *building codes*) we can reduce the vulnerability in software systems [22], [23]. Polikarpova et al. found that twice as many bugs were found when code was written with a *strong specification* [24]. Mohanani et al., however, found that specifications can lead to developers blindly following them without considering why the rules exist [25]. Our work seeks to demonstrate that the act of writing a specification creates an implicit priming effect that can impact a developer's approach to security.

B. Work on password security

There is a large body of work surrounding passwords, but a small subset that addresses how developers perform password storage and present analysis of the process. Password storage is a feature generally supported by cryptographic libraries. The usable security community has studied the developers' interaction with the cryptographic APIs.

Naiakshina et al. ran the first qualitative usability study to observe how 20 computer science students address the task of password storage [3]. They concluded that participants consider functionality before security. Unless participants are primed, they do not think the task of password storage requires a secure solution. On the other hand, participants who were primed to consider security used various hash functions and different algorithms to secure their password. For the participants who were primed, none of their solutions met the academic standards of the time. Cryptographic frameworks offer password storage as an opt-in feature. This means the developers needs to understand cryptography to store passwords. 10% of the non-primed participants attempted a secure solution for password storage while 70% of the primed participants attempted a secure solution. On asking the non-primed students about the security oversight, they replied that they would have implemented secure storage if they were writing code for a commercial product. To address this insight Naiakshina et al. conducted a field study with freelance developers. Like students, freelance developers do not consider security for password storage, unless prompted. Both students and freelance developers have misconceptions about secure password storage, however interestingly freelance developers show a wider range of these misconceptions. Freelance developers often stored passwords with Base64, confusing encoding functions with hash functions, a misconception they shared with end-users. Naiakshina et al. conclude, that even when developers believe they are coding for companies they seldom store the password securely without prompting [5]. Acar et al. conducted an experiment with GitHub developers to establish if they are an accurate representation of developers in general for security-based developer studies. The GitHub developers were asked to perform password storage securely. The solutions included the storage of plain-text passwords, use of static salts, use of unsafe hashing algorithms [26]. Our work goes beyond Acar et al.'s and Naiakshina et al.'s work by examining developer's rationale for their password storage implementations and finds that, whilst developers aren't storing passwords securely, they *think* they're following best practices.

Oesch et al. evaluated 13 popular password managers and their solutions for handling the 3 main stages of a password's life-cycle; password generation, storage, and auto-fill. Their evaluation of password storage showed that developers stored information in plain-text, left metadata unencrypted, and used insecure defaults [27]. Our work compliments this by diving deeper into *why* developers do not engage with best practice. There is a large body of work on end-user passwords and their security [28], [29], [30], [31], [32], [33], [34], [35], [36]. In contrast our work focuses on developer's approaches to storing passwords.

C. Work on secure programming

Weir et al. looked at the prevalence of security assurance techniques (including threat assessment and code review) among Android developers [37]. They found that between only 22–30% of Android developers used these techniques despite a high perceived need for security. We found that

∼56% of developers claimed to write a specification without prompting. Fischer et al. examined the amount of code copied from Stackoverflow, and its security [38]. They found that 15% of Android apps contained vulnerable code copied from Stackoverflow. We found that ∼8% of developers copied from Stackoverflow specifically, but that a further ∼12% copied from other online sources.

Many vulnerabilities arise due to developers misusing cryptographic libraries. Nadi et al. performed an empirical investigation into challenges developers face when using Java cryptographic APIs. Based on the analysis of 100 Stack Overflow posts, 100 GitHub repositories and a survey of 48 developers, they found that developers find cryptographic features such as encryption and digital signatures difficult to program. they also found that APIs are generally perceived to be too low-level for developers [39].

Egele et al. studied the integration of cryptographic APIs in Android applications. They found errors in 88% of the applications. CryptoLint was introduced as a static analysis tool to find these errors [40]. Patnaik et al. performed a thematic analysis of 2491 Stack Overflow posts from developers seeking help with using 7 cryptographic libraries, and found 16 usability issues [41] that could be related to Green and Smith's earlier work that proposes usability principles for cryptographic APIs. show that developers find cryptographic APIs challenging to use. We find that as well as struggling with APIs developers are not clear on what they need to do to store passwords securely, following current guidelines [20].

III. Method

We used a between-subjects design to explore whether the act of specification writing results in more secure code being produced.

A. Study Design

To test the effect specification writing had on implementation we designed a study where developers would implement the part of an app's code for storing passwords. We chose password storage as a task as it is security relevant, implementable within a relatively short space of time and is a common task with plenty of guidance available that most developers would have encountered in their work.

Our study was implemented as a set of online tasks and questions (to capture rationale). Developers were randomly assigned a grouping (either *specification* or *no-specification*) and shown the following scenario:

> You are working on the backend of an application. Users create an account on the app, and login before being allowed to use the program. The application is complete bar one task: writing the login system users use to authenticate with the app. You have been tasked with implementing this part of the app. You decide to start with storing the users' passwords. Your boss trusts your judgment when it comes to implementing this feature.

Developers in the *specification* group were then asked to write a specification for how the password should be stored.

> You decide to start by writing a specification for how the password should be stored, and to note down any special requirements and implementation details. You are provided with a username and password, and they have been checked to see that they are valid text.
>
> Describe your specification below. You can describe your specification using formal notation, informal notes, a list, mathematical notation or any other method. If you draw a picture as part of your specification, please say so and say what is shown.

Both groups were then asked to implement the password storage using whichever language they wished. If they used a real programming language they were asked to note it.

> You start writing the password storage method. You have been given the password the user wishes to use and you need to store it so that it can be checked whenever users try and login. You are given a username and password. Both have been checked to be valid text (i.e. neither empty nor containing bad characters) Write code (or pseudocode) to implement the password storage. Your code doesn't need to be compilable or syntactically correct but should illustrate your general approach to the implementation.

Developers in the *no-specification* group were then asked if they had made some form of specification or plan before starting their implementation (without being asked to). Those that indicated that they did, were asked to describe their specification and their results were added to those of the *specification* group. Both groups were then asked to provide a rational for their coding approach in a free text box. They were asked if they considered *what threats might attack a stored password*, and, whether they *referred to any standards for password storage* when implementing the code. Finally, participants were asked whether they had any formal qualifications in software engineering, or computer science; and, to rate their knowledge of security and cryptography on a 5-point Likert scale,and briefly describe their security and development experience.

B. Analysis

To analyze the data we scored each of their implementations using Naiakshina's metric [3] and compared the average score between different groups using the Mann Whitney U test (a rank-based non-parametric test to explore if two groups are distinct [42]) to test for significance and to calculate the effect sizes (using the rank-biserial correlation [43]). To analyze developer's rationale and threat models we asked developers to describe them and analyzed them qualitatively using a grounded theory approach [44], [45].

C. Recruitment and Ethics

Developers were recruited from *Prolific Academic* and were screened, by Prolific, based on their familiarity with computer

programming. Developers were offered a financial reward for completing the study of $X*, inline with the *living wage* in our country. All developers who completed the study were paid for their work.

Ethical approval for the study was sought from and granted by XXX* University. No personal data was collected, and demographic data was deleted after coding and validation. Data is available online at: XXX*.

D. Limitations and Threats to Validity

We acknowledge the following limitations and threats to our study:

- Our developers were recruited by *Prolific Academic* and as such, may not be representative of how developers as a whole behave. Other studies have also used similar populations for studying passwords and developers [46], [31], [47].
- Developers may not know how to store a password, and may not be aware that it is a security related task. We mitigate this by qualitatively analyzing the developers' rationale behind their code.
- Developers who were not prompted to write a specification, may opt to write a specification anyway. To correct for this we asked developers not in the specification group if they wrote a specification, after their implementation. We assume that the specification produced by the unprompted group is similar to the prompted group (and we ask them to describe it), but this may not be the case and some participants may retrospectively write a specification.
- We ask the developers about their qualifications and experience, however all data is self-reported and may not be accurate.
- We asked developers to implement password storage and 99 developers (72%) did so. 19 developers (14%) instead appeared to write code implementing password authentication (how one would check if a password was correct) but from which their approach to password storage could be seen. A further 15 developers (11%) stored the password, but did so only checking if the password contained a suitable range of letters, numbers and symbols, 3 (3%) approached the problem by retransmitting their passwords over HTTP[1], and 1 insisted the passwords be stored *alphabetically*. We include all in our analysis, as they were all conceivably ways a developer may approach storing passwords.
- Scoring implementations according to Naiakshina's criteria could introduce subjectivity. To mitigate this, one author scored and then another author independently rescored all the implementations and calculated Cohen's Kappa (a measure of inter-rater reliability [48]). The kappa-value indicates *almost perfect agreement* ($\kappa = 0.94$) [48]. Similarly, our codebooks, whilst grounded in data, were likely influenced by the coder's background and experiences. Using our codebooks a separate coder independently re-coded the entire dataset. We found *substantial agreement* ($\kappa = 0.72$) with our coding for developers' explanations for their implementations (Table VI) and *almost perfect agreement* ($\kappa = 0.84$) with our coding for the threats developers considered.
- We measure developers' password storage approaches using Naiakshina's criteria, but this poses a construct validity threat. We chose this metric as it has been used in prior work [3], [5], and on a NIST standard for password storage [20]. We mitigate this threat by qualitatively analyzing *why* developers wrote the code they did as well as their implementations.

*Redacted for blinding

[1]Three appeared to have copied the question from: https://stackoverflow.com/questions/19999417/password-storage-in-code-how-to-make-it-safe.

IV. Quantitative Results

Table III reports how the teams scored against Naiakshina's criteria (Figure 1). In our sample, only 53 developers (38%) produced outputs that fulfilled at least one part of Naiakshina's criteria. The most common criterion fulfilled was that of hashing data (demonstrated by 38% of participants who scored a point, 14% of overall sample). Just under 20% of the developers who scored a point used a random salt or an appropriate hash length (7% overall); and the remainder of the points in Naiakshina's criteria were awarded rarely.

A. RQ1: Do specifications lead to securely stored passwords?

Developers prompted for a specification ($n = 61$) scored better ($\mu = 1.03$) than those that were unprompted ($n = 77$, $\mu = 0.47$)—a comparison by Mann-Whitney U suggests that this is a significant difference ($p = 0.024$, $U = 1947.5$), but with only a small effect size (rank-biserial coefficient [43], $r_{rb} = 0.171$). There remains a significant difference in performance if we omit the subset of the unprompted group who reported writing a specification without being asked to—prompted participants ($n = 61$, $\mu = 1.03$), in contrast to developers who did not write a specification ($n = 34$, $\mu = 0.38$). The two groups are distinct (Mann-Whitney $U = 820$, $p = 0.027$) but the effect size remains small ($r_{rb} = 0.209$). However, a comparison between all participants who wrote a specification, prompted or not, ($n = 104$, $\mu = 0.83$) and those who did not write a specification ($n = 34$, $\mu = 0.38$) is not statistically significant ($p = 0.061$, $U = 1495$, $r_{rb} = 0.154$). This could be explained by developers in the *unprompted specification* group (those who were not asked to write a spec but who claimed to have written one anyway) actually writing their spec after their implementation in response to us asking if they had written one beforehand. This theory is supported by Table II where we found no significant difference between the unprompted specification and the group that claimed not to write a specification ($p = 0.247$).

The distribution of scores is given in Table I. 50–70% of developers did not store a password in any meaningfully secure way (a score of 0), and no developer obtained a perfect score (of 7) using Naiakshina's metric, although two developers did meet the score indicating best practice (a score of 6; both were in the *prompted specification* group). Of the 77 developers whom we did not prompt to write a specification 56% (43) claimed to write one anyway unprompted; 26% (36)

TABLE I

DISTRIBUTION OF SCORES FOR PASSWORD STORAGE METHODS BY DIFFERENT GROUPS. ABSOLUTE VALUES ARE GIVEN IN (PARENTHESES). THE SPECIFICATION GROUP CONSISTS OF TWO-SUBGROUPS: THOSE THAT WE EXPLICITLY PROMPTED FOR A SPECIFICATION, AND THOSE THAT WE DID NOT PROMPT BUT REPORTED WRITING ONE UNPROMPTED. A SCORE OF 6 OR MORE IS CONSIDERED TO BE FOLLOWING BEST PRACTICE.

Group	Count	> 0	0	1	2	3	4	5	6	7	μ	σ
Specification	104	43	59% (61)	20% (21)	11% (11)	5% (5)	4% (4)	0	2% (2)	0	0.83	1.30
Prompted specification	61	27	56% (34)	15% (9)	15% (9)	7% (4)	5% (3)	0	3% (2)	0	1.03	1.51
Unprompted specification	43	16	63% (27)	28% (12)	5% (2)	2% (1)	2% (1)	0	0	0	0.53	0.88
Unprompted	77	26	66% (51)	25% (19)	6% (5)	1% (1)	1% (1)	0	0	0	0.47	0.79
No Specification	34	10	71% (24)	21% (7)	9% (3)	0	0	0	0	0	0.38	0.65
Used standard	36	16	56% (20)	19% (7)	11% (4)	8% (3)	3% (1)	0	3% (1)	0	0.94	1.41
No standard	102	37	64% (65)	21% (21)	10% (10)	2% (2)	3% (3)	0	1% (1)	0	0.64	1.10
Formally qualified	59	27	54% (32)	22% (13)	15% (9)	5% (3)	3% (2)	0	0	0	0.81	1.09
Not formally qualified	79	26	67% (53)	19% (15)	6% (5)	3% (2)	3% (2)	0	3% (2)	0	0.65	1.26
Overall	138	53	62% (85)	20% (28)	10% (14)	4% (5)	3% (4)	0	1% (2)	0	0.72	1.19

TABLE II

COMPARISON BETWEEN GROUPS USING THE MANN-WHITNEY U TEST.

Group 1	Group 2	U	p	r_{rb}
Prompted Specification	Unprompted	1948	0.024	0.171
Specification (all)	No specification	1495	0.061	0.154
Prompted specification	No specification	820	0.027	0.209
Unprompted specification	No specification	675	0.247	0.077
Used standard	No standard	1638	0.135	0.108
Formal qualification	No qualification	2018	0.061	0.134

TABLE III

FREQUENCY DIFFERENT POINTS IN NAIAKSHINA'S CRITERIA WERE OBSERVED COMPARED TO THE WHOLE POPULATION. NO ANSWER SCORED A HALF-POINT FOR KEY-STRETCHING. (ABSOLUTE VALUES).

Criteria	Observations	
Hashed	38%	(53)
Salted	14%	(19)
Hash length	7%	(9)
Key stretching	2%	(3)
Memory-hard hashing	1%	(1)
Random salt	7%	(10)
Salt length	3%	(4)

TABLE IV

CO-OCCURRENCES OF POINTS IN NAIAKSHINA'S CRITERIA (I.E. 36% OF ALL PARTICIPANTS WHO HASHED THEIR PASSWORD ALSO SALTED THEIR PASSWORDS). (ABSOLUTE VALUES).

	Hashed	Salted	Hash length	Key stretching	Memory-hard	Random salt	Salt length
Hashed		36% (19)	17% (9)	6% (3)	2% (1)	19% (10)	8% (4)
Salted	100% (19)		26% (5)	11% (2)	0% (0)	53% (10)	21% (4)
Hash length	100% (9)	56% (5)		22% (2)	0% (0)	56% (5)	22% (2)
Key stretching	100% (3)	67% (2)	67% (2)		0% (0)	67% (2)	67% (2)
Memory-hard	100% (1)	0% (0)	0% (0)	0% (0)		0% (0)	0% (0)
Random salt	100% (10)	100% (10)	50% (5)	20% (2)	0% (0)		30% (3)
Salt length	100% (4)	100% (4)	50% (2)	50% (2)	0% (0)	75% (3)	

of developers reported referring to some kind of standard or guide when writing their password storage method; 43% (59) claimed some formal software engineering qualification.

Finding: Examining the rank-biserial correlation (r_{rb}) to the scores themselves in Table I, suggests that whilst forcing developers to write a specification before coding will lead to more secure password storage approaches ($p = 0.024$), it isn't going to make a huge difference—developers *might* remember to hash them or to add salt, but will still leave them stored insecurely.

B. So what else has an effect?

If the act of forcing developers to write a specification only has a small impact on their ability to store passwords securely, then do we find anything else having an effect?

TABLE V

OBSERVATIONS OF SPECIFIC HASHING METHODS USED BY DEVELOPERS. SOME DEVELOPERS RECOMMENDED MULTIPLE HASHING METHODS.

Hash	Observations
Encryption	9
AES	1
MD5	6
SHA1	4
SHA256	7
SHA512	1
base64	3
Custom cipher	1
bcrypt	6
PBKDF2	4
Argon2	1
Insecure method	26
Secure method	11

Participants reported their familiarity with cryptography on a 5-point Likert scale. There is a small positive relationship between reported cryptography experience and score (Spearmans Rho [49], $\rho_s = 0.333$, $p = 0.00$), with most developers reporting that they had *little to no experience* (107, 78%) (*No experience*: 44 (32%), *little experience*: 63 (46%), *moderate experience*: 25 (18%), *very experienced*: 4 (3%), *extremely experienced*: 2 (1%)). This is in contrast to the findings of Hazhirpasand et al. who found no significant relationship between developer experience and their ability to use a cryptography API [50]—though Hazhirpasand et al. rated developer experience on the basis of activity on GitHub, as opposed to a self-reported value. We did not find a significant relationship between developers who had a formal software engineering qualification and those who did not ($p = 0.061$). Participants who reported using a standard to inform their code implementation scored better than those who used no standard but not significantly ($p = 0.135$).

C. RQ2: What did developers do?

Our observations of hashing and salting rates are broadly inline with what Naiakshina et al. observed [3], where an overall 35% ($\frac{7}{20}$) of developers hashed passwords and 25% ($\frac{5}{20}$) also salted them—however Naiakshina et al.'s study explicitly primed half of their developers ($\frac{10}{20}$) by asking them to store them security, and only the primed groups hashed or salted their passwords. In contrast, in our findings we observe similar rates over all participants.

In our study we asked developers to provide code in any programming language, including pseudocode. Most developers described their implementation in these terms using functions called `hash` and appending salts, however some gave specific methods for storing their passwords. Table V shows the specific methods we encountered for hashing passwords. Many developers recommended hash functions that were inappropriate for password storage[2]—including MD5, the SHA family, and a substitution cipher. Other developers recommended encryption (which is unsuitable for password storage [20]), or even using base64. Of all the developers who stored their passwords hashed, 50% (26) used an inappropriate hashing method [20], and only 21% (11) recommended a secure modern password hash. One developer recommended both a secure and insecure method:

```
...hash password in bcrypt or md5...
```
(scored 1)

Finding: Only a third of developers wrote code to store their passwords hashed. 50% of those developers recommended an insecure hash function, and only 21% recommended a secure hash function. The remaining 29% did not specify the method—they just 'hashed' them. 14% of developers remembered to salt and hash their passwords. More comprehensive security (Figure 1) was rare.

[2]They are quick to calculate using little memory, thus making them amenable to cracking, unlike memory hard hashes such as PBKDF2.

V. RQ3: Why are developers storing passwords like this?

After implementing their solutions we asked developers why they had used a particular approach. Two of the authors used a grounded theory approach [44] to analyze the responses. Two passes were required to reach the point of theoretical saturation [45] when no new codes were identified. The resulting codebook, and illustrative examples of each code, is shown in Table VI. We also contrasted the distribution of codes in the *prompted specification* and *no specification* groups and found them to be broadly similar—with the *no specification* group being slightly more likely to report they wrote their code the way they did because the implementation was easy. Consequently our remaining analysis of developers explores why they implemented password storage in the way they did, and is over the entire study group.

We also asked the developers if they considered any threats when implementing their password storage solution? Threat modeling is a standard technique when designing for security [51] that encourages developers to consider what defenses are needed to mitigate the potential threats to a system. 55% (77) developers reported considering potential threats when implementing their password storage solution. Their responses were analyzed by one author, again using a grounded theory approach [44]. Two passes were required to reach the point of theoretical saturation [45] when no new codes were identified. The resulting codebook, again with illustrative examples of each code, is shown in Table VII.

A. You either think you do know, or you know you don't know

Our analysis of the reasons why developers implemented password storage in the ways they did reveal two interesting sub-groups. Several answers appear to indicate that the developers thought they had stored the passwords properly (the *experience*, *replication of previous efforts*, *perceived best practice* and *taken under advisement* codes); whereas others seemed to know that their implementation was limited and that they didn't know how to do it (the *naïve*, *acknowledgment of limitations* and *only way I know* codes).

Within the group who thought they knew how to store passwords, developers indicated that they believed their approaches were *best practice*. One developer stored the password directly into a database:

```
INSERT INTO `users` (`username`, `password
    `) VALUES ('user', PASSWORD('password1
    '));...
```

They explained this as:

"Because that is the best way to store the password"

Yet their solution stores the passwords directly without hashing or salting: they scored 0. Others stated:

"this is most accepted way of storing passwords" (scored 4)

"It's based on corporate best practices" (scored 0)

TABLE VI

CODEBOOK FORMED FROM THE ANALYSIS OF DEVELOPERS' EXPLANATION OF THEIR IMPLEMENTATION APPROACH. QUOTES ARE GIVEN TO ILLUSTRATE THE USE OF ALL CODES WITH RELEVANT PASSAGES UNDERLINED. SOME RESPONSES WERE ASSIGNED MORE THAN ONE CODE. NO MORE THAN 3 CODES WERE USED TO CAPTURE ANY SINGLE RESPONSE.

Code	Description	Count	Prompted Spec	No Spec
Implementation Ease	The developer wrote it like that as the implementation would be *"simple"*	36	16%	29%
	"Because it was a simple but quite effective way to store. To ensure that the data is secure, the function that encrypts the password must be very good." (scored 0)			
Readability	The developer focused on how understandable their code would be to a reader.	8	7%	9%
	"I wrote that way because it shows the idea very clearly. The encryption code is a more difficult question and needs time and ideas to implement a good encryption." (scored 0)			
Naïve	The developer wrote the code in a literal manner without considering the merits of any other approaches.	13	7%	12%
	"Because I don't know how to wrote code, so just used a literal approach." (scored 0)			
Experience	The developer made reference to their experience when describing how they wrote their code.	19	11%	18%
	"I'm somewhat experienced in applied security and I consider the password should be stored securely, considering the worst case possible." (scored 3)			
Replication of previous efforts	The developer said they had done it like this before.	8	5%	6%
	"I wrote code like this because it is something I have done before. I've written a login system for a password manager so recognise that passwords before storage should always be hashed or encrypted to avoid storing them in plain text. I used a struct mainly for storage purposes of this task, but would normally use a database such as SQL to store them, after hashing." (scored 2)			
Feature justification	The developer justified a specific feature of their implementation (e.g. ability to send password reminders).	4	3%	3%
	"This is a simple way to code and allow for a reminder to the recipient!" (scored 0)			
Method justification	The developer justified the structure of their code.	20	18%	15%
	"I used a utility class. This class stores usernames and passwords in a Map data structure, and then provides functions for user registration and login." (scored 0)			
Acknowledgment of limitations	The developer noted that their code has limitations, and that it doesn't have a certain feature (e.g. it is insecure).	14	7%	12%
	"It assures the storage of password and allows to recover the password easily even if the security is not high." (scored 3)			
Perceived best practice	The developer did it this way as this is the correct way to store a password or a standard way in their company.	20	18%	12%
	"this is most accepted way of storing passwords" (scored 4)			
Consideration of threats	The developer considered a threat that might attack the code and explicitly attempted to mitigate that threat.	18	18%	9%
	"Hashing passwords is a necessity, storing passwords in plain text is a huge security concern: and should never even be considered." (scored 2)			
Taken under advisement	Someone told them this was a good way to do it.	1	0%	0%
	"my friend who is into cybersecurity told me about this" (scored 1)			
Only way I know	The developer indicates that this is the only way they knew how to complete the task.	15	11%	12%
	"That was the only way i knew to solve that problem" (scored 0)			

TABLE VII

CODEBOOK FROM THE ANALYSIS OF DEVELOPERS' RESPONSES TO WHAT THREATS DID THEY CONSIDER WHEN STORING THE PASSWORDS. ONLY DEVELOPERS WHO INDICATED THAT THEY HAD CONSIDERED A THREAT'S RESPONSES WERE ANALYZED. QUOTES ARE GIVEN TO ILLUSTRATE THE USE OF ALL CODES WITH RELEVANT PASSAGES UNDERLINED. SOME RESPONSES WERE ASSIGNED MORE THAN ONE CODE. NO MORE THAN 4 CODES WERE USED TO CAPTURE ANY SINGLE RESPONSE.

Code	Description	Count	Prompted Spec	No Spec
Access	Threat from unauthorized access to the database (e.g. leaks).	36	34%	15%
	"Password leaks" (scored 6)			
Cracking	Threat from attacks on stored passwords (e.g. cracking or rainbow tables).	20	16%	12%
	"reverse the hash code but i think that is impossible because is unidirectional" (scored 1)			
Hacking	Threat from unspecified threat actors, phishing or social engineering.	16	11%	12%
	"Stealing them by a hacker, hacked by an unknown user to steal information and data" (scored 0)			
Programming concerns.	The threat from vulnerabilities in their code (e.g. bugs, SQL injection)	10	8%	3%
	"SQL injection, unauthorized DB access" (scored 3)			
Confidentiality	Concerns about making the stored passwords harder to see.	10	10%	3%
	"Someone accessing the content that is not the main user. If I used strings the password would be stored in strings until the Garbage Collector clears it and we cannot control when that happens." (scored 0)			
Malware	Threat from malware, key-loggers or network attacks.	6	5%	6%
	"Someone tracing you with keylogger or maybe trojan horse" (scored 2)			
Reflection	Consideration of what they *should* have done and the security of their implementation.	5	3%	3%
	"Since the good is quite simple, I am not certain if the storage is secure." (scored 0)			
Wider-context	Concerns about the wider impact of an insecurely stored password.	2	2%	0%
	"Potential of a database dump, hackers can just login if the passwords were stored in plain text, with hashed passwords they would need to brute force the password. Failure to secure our users passwords could lead to them having their accounts on other platforms compromised too as users tend to reuse passwords." (scored 2)			
Insider-threat	Threats from insiders who might have access to stored passwords.	2	3%	0%
	"The database being accessed by a 3rd party, internal threat actors(excepting those with access to the code for password storage)" (scored 4)			

Developers indicated that they knew their answer was correct because they had done similar tasks before calling upon both their experience as well as previously written code:

> *"Because I wrote a user registration system in the past."* (scored 2)
>
> *"I'm used to implementing similar login and authentication mechanisms in university projects and the thought process is always the same:..."* (scored 0)
>
> *"I wrote code like this because it is something I have done before. I've written a login system for a password manager so recognize that passwords before storage should always be hashed or encrypted to avoid storing them in plain text...."* (scored 2)

The relevant part of the code based on the login system for the password manager looked like:

```
user.username = std::cin.get();
user.password = hashPassword(std::cin.get
    ());} ...
void hashPassword(std::string password) {
 //Cryptography algorithm to hash password
    , preferably using a salt }
```

It hashed the password with a *cryptography algorithm*. It would *preferably* use salt. Another developer told us that they had taken advice from someone they considered knowledgeable about cybersecurity:

"my friend who is into cybersecurity told me about this" (scored 1)

Yet the solution their friend supplied was mostly inadequate, only showing signs of hashing (with MD5).

```
... string hashpass = MD5(password);
PasswordDatabase.put("login","password");"
```

This group of developers appear to believe their answers are correct, and that they are following best practice. They indicate that code similar to what they wrote is in projects they've implemented. Yet despite this, there are many low scores. One developer described their experience as:

"I have been working as a software engineer for 8 years and have developed authentication systems for our clients hundreds of times so have come to learn the best practices for doing so." (scored 3)

Their score would suggest they have more to learn.

Not all developers seemed to be so confident. In contrast to the first group, other developers gave explanations that suggest they are aware that they don't know how to store passwords properly, or at least that their code had limitations. For example one developer stated:

"I don't have a lot of knowledge about password storaging..." (scored 1)

Their implementation hashed the password, suggesting the developer was confused about the distinction between *hashing* and *encryption*:

```
... string encpass = anHashingFunction(
    password);
myfile << username << endl;
myfile << encpass << endl;
```

They scored 1 point using Naiakshina's criteria, for hashing the password. Another said:

"I wrote the code that way since it's the only way I know how to check if the passwords are valid, and the hashing / storing bit because unhashed passwords are unsafe. ... Other methods could be used to encrypt the password, but I've heard hashing or MD5 hashing is the most common." (scored 1)

The following two explanations came from developers who were in the top 10% of highest scoring implementations, according to Naiakshina's criteria. Both acknowledge the security of their implementations, and that it wasn't perfect:

"It assures the storage of password and allows to recover the password easily even if the security is not high." (scored 3)

"It was the simplest and easiest way I could think ... This way it protects most cases, but of course a more elaborate with more defence lines is needed (and the salt implementations is not very well done, ...)" (scored 4)

This group of developers form a counterpoint to the first group who think they know how to store passwords: they know they don't know everything. Whilst directly comparing groups is hard (the codes are not independent, and emerge from what developers said) a comparison of average score between them suggests that neither group is storing passwords more securely (comparison of mean score between the *think they know* and *know they don't know* groups: 0.79 vs 0.63). In short, roughly a third of developers appear to be overly confident in their knowledge of best practice in our study. Despite this their answers do indicate that developers are aware that password storage is an inherently security oriented task. They know they should be storing passwords securely, but plenty of them are overconfident and have misplaced assurance in what they do.

B. On reflection, perhaps you know

The *acknowledgment of limitations* code from Table VI and the *reflection* code from Table VII are interesting as they highlight when developers indicated that their implementation was lacking security aspects. For example, one developer remembered to use a hash function with a suitable length in their implementation. When stating which threats they considered they note that they had forgotten to salt the password (and why that was necessary).

"...(Thinking about it, it might have been a good idea to concatenate some constant text at the end of the password so that whether the user uses the same password on two different attacked services cannot be determined simply by checking whether the hashes are identical.)" (scored 2)

Others, on reflection, realized their solution was inadequate (with respect to security). Both of the following developers stored their passwords directly as plain-text (scoring 0), yet when asked to consider the threats they later seemed to realize that there were some they should have considered:

"Unfortunately, I have not considered any threats, but I know that the password should be encrypted." (scored 0)

"Actually I didn't consider them in the pseudo-code but I assume there are some threats like brute hacking" (scored 0)

One developer stored their password directly, but when considering threats gave a guide to storing them that would have scored at least 3:

"The best security practice is not to store the password at all (not even encrypted), but to store the salted hash (with a unique salt per password) of the encrypted password." (scored 0)

Whilst we did not prime developers for storing passwords securely as Naiakshina did [5], we still found that developers talked about the security of their implementation. Around half of the developers reported considering threats when implementing their solutions, and some made reference to those threats when describing why they'd implemented the code in the way that they did. The threats described in Table VII are reasonable: the reason we hash and salt passwords is to ensure if the database is *accessed* illegally, that the passwords cannot be trivially *cracked*. This is hopeful: it suggests developers may be learning that passwords and security are linked and that they should store them securely, not *if we want* them to [5]. Developers may not realize it immediately that passwords

TABLE VIII
SITES APPARENTLY REFERENCED BY DEVELOPERS TO IMPLEMENT PASSWORD STORAGE.

Source	Count
stackoverflow.com	12
gist.github.com	1
docs.microsoft.com	1
simplecode121.blogspot.com	1
www.programcreek.com	1
happycoding.io	1
www.baeldung.com	1
pypi.org	1
www.codota.com	1
medium.com	1
www.the-art-of-web.com	1
www.tutorialspoint.com	1
www.cpp.re	1
docs.python.org	1
www.w3resource.com	1
howtodoinjava.com	1

should be stored securely, but if given time to reflect they do seem to make that connection. Even if a developer doesn't initially realize that password storage is a security oriented task, by giving them time to reflect (in complement with time to consider a specification) some developers do realize that passwords must be stored securely.

C. Google is your friend

There is much guidance and advice online about how to store passwords (on sites such as *Stack Overflow*, for example). When analyzing developers reasons and implementations, we checked for copying from such sites. We searched online to see if any of the implementations and pseudocode developers provided appeared online and found 12 appearing on the *Stack Overflow* developer forum alongside 15 others appearing on other websites (Table VIII). We found that 27 developers (20%) appeared to have copied code from various sites (shown in Table VIII), with the majority having taken code directly from Stack Overflow. Of these 27, 7 reported using a standard.

The group that used the online source appeared to score significantly higher than the group whose source was unknown ($\mu = 1.19$ vs $\mu = 0.60$, $p = 0.017$) though the effect size was relatively small ($U = 1155$, $r_{rb} = 0.23$); however when reading the solutions online we noticed that some of the articles developers appeared to have copied from also contained guidance on how to store them near-perfectly (according to Naiakshina's criteria). One developer justified their answer as following:

> *"I wrote the code like this because it is good practice not to store a password in clear."* (scored 2)

The solution appeared to have been taken from a *how-to* site which described how to implement password storage with a variety of hashes and salts, starting with MD5 and ending with bcrypt and scrypt; however the site went on to describe a solution at the end of the article that would have scored 6 points (losing the last point for only using 16 instead of 32 bits for the salt).

Another developer described their implementation as:

> *"The first and foremost way to store passwords in your database is to have the plain text.. . . "* (scored 0)

This came from a blog post [52]. The remainder states:

> *"(don't do this) I can't emphasize strongly enough that you should NEVER, EVER, store passwords in plain text."*

The article does describe how to store passwords using a hash and a randomly generated salt (3 points); yet again the developer only copied the insecure counter-example at the start of the article.If we want developers to store passwords correctly then we need to make sure the code we want them to copy is immediately obvious. That developers are copying code from online isn't of itself worrying—if they copied *the right* solutions we might see secure password storage. Instead, some developers seem to be copying online code, using the articles to justify themselves, but not reading the article all the way through.

VI. DISCUSSION

A. Do specifications lead to securely stored passwords?

Writing a specification has a small, but positive, effect on developers ability to store passwords securely. Yet in saying this, we avoid the bigger issue that developers seem to really struggle with implementing password storage correctly. In our study 62% of developers failed to hash, salt or add any security mechanism whatsoever.

Our paper joins an ever-growing body of work demonstrating that developers are struggling implementing password storage [53], [3], [5]; but our work also finds that it isn't *just* that developers struggle to use cryptography APIs [54], [41]; and it isn't *just* that developers don't know enough about cryptography to complete the task correctly: developers stated that hashing passwords with MD5 was best practice (it isn't [20]). Developers would forget about salting and say that's the way that they do it in their company. They would claim cybersecurity expertise, to have password storage code in production, as reasons why their code is secure; as reasons why their code follows best practice—and yet they fall short. Our paper finds that specification is beneficial ($p = 0.024$), but, equally importantly, it highlights that developers don't know that they're storing passwords insecurely.

B. Beyond Naiakshina's criteria

In this study we measured developers' ability to store passwords using Naiakshina's criteria, as a proxy for the NIST SP 800-63-3 standard [20] which defines current best practice. The criteria and standard itself is somewhat quiz-like, asking developers to remember cryptographic techniques like hashing and salting as well as arbitrary lengths and counts. If developers do not know these requirements then they will not remember them. so what then do we learn *beyond* the fact that developers do not seem to recall Naiakshina's criteria?

An ideal specification might have listed the criteria in full as functional requirements, but it might also have been as simple

as: *"store the password securely, following NIST SP 800-63-3."*; yet none of the developers in our study made reference to any standard in their specifications. Developers seemed to know there was a *best practice* they ought to be following, yet didn't appear to go look up what it actually was. Whether this generalizes and developers' recollection of other standards is equally poor is a topic for future work.

Whilst one good approach to implementing password storage is to do what the standard says, another equally valid (and arguably better) aproach is to use a framework and let it do it for you. Web-frameworks, like Django, include password storage systems (and in Django's case explicitly reference NIST password standards[3]) and can take care of passwords for developers. Again, developers in our study did not appear to make use of frameworks like this, so is it that developers are unaware of these features inside frameworks, or did they choose not to use them?

Perhaps given the seeming recalcitrance towards reading standards, the resistance towards using frameworks, and the confidence many displayed that they were in fact following best practice we might conclude that developers are over confident in their abilities. Why use a library when you can implement it yourself trivially? Why check the standard when you know already what best practice is? Developers seem to have learned not to *roll their own crypto*; perhaps they should also consider avoiding *rolling their own authentication* in future too?

C. Developers are still not the enemy

We say that *users and developers are not the enemy* [55], [54]—that we must not blame users or developers when an API or security interface is not designed for a human to be able to use correctly. Yet when we talk about password storage we present it as a list diktats that developers *must* implement to ensure they do the task correctly. As we, and others [53], [3], [5], have shown developers cannot follow these instructions. Perhaps then, instead of pointing out that developers can't store passwords and providing lists on what they *must* do, we should *fit the task to the human* and provide alternative mechanisms for storing passwords correctly without having to remember what the current best practice actually is, or understand the intricacies of various hashing schemes. Truman said of being the President: *"He'll sit here and he'll say, "Do this! Do that!" And nothing will happen"* [1]; and a comparison can be drawn to the security and cryptography communities: we cannot keep sitting here; saying, *"Hash this! Salt That!"* and pointing at NIST SP 800-63-3, because Truman was right: *"nothing will happen"*. We need to find usable mechanisms for password storage.

What might these mechanisms look like? Cryptography libraries, such as Google's *Tink* [56], are attempting to wrap cryptographic details so that developers can use cryptography without understanding what a hash really is [57]. There has been limited usability validation of such approaches [58], however, and further work, documentation, and exemplar code [41] is needed to show whether this approach is effective. Alternatively, some developers seemed to copy code from online sources—ensuring that developers can find the trivially find the *right* way to store passwords and that the *right* code is trivially available may also help developers without requiring them to understand the cryptographic details.

Finally, a different solution altogether may be to encourage developers not to store passwords at all and instead use federated identity management systems (such as OAuth [10]). Whilst these systems can remove the need for some app developers to implement cryptography correctly, they come with their own set of privacy and security *gotchas* [59] and challenges [60]—we should be cautious that by recommending an alternative to passwords we are not replacing the challenge of storing a password with the challenge of implementing a federated authentication system. Work on privacy-preserving federated identity management has helped to resolve some of the privacy challenges associated with federated identity management [61], [62], though these are yet to be widely adopted in practice.

Explicitly prompting developers to write a specification does help improve the quality of password storage; but developers are still mostly failing at password storage whilst still believing they are getting it right. Giving developers time to reflect helps them realize the limitations of their approach: but until we have *developer-centered usable* password storage methods, the problem of poorly stored passwords isn't going away. We can do better than saying *"Do this! Do that!"* and watching nothing happen.

VII. Conclusion

Does the act of writing a specification (how the code should behave) for a piece of security sensitive code lead to developers producing more secure code? In a statistical sense: yes, though the effect is small ($p = 0.027$, $r_{rb} = 0.209$). In a broader sense however we show that whilst writing a specification does help developers remember more of the conditions for secure password storage, leaving this task to a memory exercise and hoping developers refer to a standard isn't working.

Future work should examine and empirically evaluate alternative strategies for helping developers complete authentication tasks—whether in the form of usable cryptography libraries, privacy preserving federated identity schemes, or alternative awareness schemes to diktats and standards. Additionally, whilst this study looked to see if *any* form of specification improved developers ability to store passwords correctly; *specific* approaches (whether that be software building codes [63], requirements engineering, or formal verification) may yield more promising results. Finally, in this study we saw developers struggling to remember how to do secure password storage, but we may see similar results for other areas where knowledge of what the *right thing to do* is conveyed only through diktats and standards—future work should examine whether this result is general or specific to password storage.

[3]https://docs.djangoproject.com/en/3.1/topics/auth/passwords/

References

[1] R. E. Neustadt, *Presidential power*. New American Library New York, 1960.

[2] F. P. Brooks Jr., *The Mythical Man-Month*. Addison Wesley, 1975, ch. 6. Passing the Word.

[3] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why do developers get password storage wrong?: A qualitative usability study," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM. Dallas, TX: ACM, 2017, pp. 311–328.

[4] A. Naiakshina, A. Danilova, C. Tiefenau, and M. Smith, "Deception task design in developer password studies: Exploring a student sample," in *Fourteenth Symposium on Usable Privacy and Security*, M. E. Zurko and H. R. Lipford, Eds. Baltimore, MD: USENIX Association, 2018, pp. 297–313. [Online]. Available: https://www.usenix.org/conference/soups2018/presentation/naiakshina

[5] A. Naiakshina, A. Danilova, E. Gerlitz, E. von Zezschwitz, and M. Smith, ""If you want, I can store the encrypted password": A password-storage field study with freelance developers," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, S. A. Brewster, G. Fitzpatrick, A. L. Cox, and V. Kostakos, Eds. Glasgow, Scotland: ACM, 2019, p. 140. [Online]. Available: https://doi.org/10.1145/3290605.3300370

[6] A. N. Meyer, T. Zimmermann, and T. Fritz, "Characterizing software developers by perceptions of productivity," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, A. Bener, B. Turhan, and S. Biffl, Eds. Toronto, ON, Canada: IEEE Computer Society, 2017, pp. 105–110. [Online]. Available: https://doi.org/10.1109/ESEM.2017.17

[7] D. van der Linden, P. Anthonysamy, B. Nuseibeh, T. T. Tun, M. Petre, M. Levine, J. Towse, and A. Rashid, "Schrödinger's security: Opening the box on app developers' security rationale," in *42nd International Conference on Software Engineering (ICSE)*, 2020.

[8] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? An analysis of topics and trends in Stack Overflow," *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, 2014.

[9] X.-L. Yang, D. Lo, X. Xia, Z.-Y. Wan, and J.-L. Sun, "What security questions do developers ask? A large-scale study of Stack Overflow posts," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 910–924, 2016.

[10] D. Hardt *et al.*, "The OAuth 2.0 authorization framework," RFC 6749, Tech. Rep., 2012.

[11] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Software*, vol. 1, no. 1, p. 75, 1984.

[12] D. L. Parnas, "A technique for software module specification with examples," *Communications of the ACM*, vol. 15, no. 5, pp. 330–336, 1972.

[13] J. Spolsky, "The Joel test: 12 steps to better code," in *Joel on Software*. Springer, 2004, pp. 17–30.

[14] T. DeMarco, "Structure analysis and system specification," in *Pioneers and Their Contributions to Software Engineering*. Springer, 1979, pp. 255–288.

[15] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992. [Online]. Available: https://doi.org/10.1109/2.161279

[16] S. Furnell and K.-L. Thomson, "Recognising and addressing 'security fatigue'," *Computer Fraud & Security*, vol. 2009, no. 11, pp. 7–11, 2009.

[17] S. Parkin, K. Krol, I. Becker, and M. A. Sasse, "Applying cognitive control modes to identify security fatigue hotspots," in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/soups2016/workshop-program/wsf/presentation/parkin

[18] A. van Lamsweerde, "Formal specification: a roadmap," in *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*, A. Finkelstein, Ed. ACM, 2000, pp. 147–159. [Online]. Available: https://doi.org/10.1145/336512.336546

[19] A. Padegs, "System/360 and beyond," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 377–390, 1981.

[20] P. A. Grassi, M. E. Garcia, and J. L. Fenton, "Digital identity guidelines," NIST, Standard Special Publication 800-63-3, 2017.

[21] R. G. Dromey, "Cornering the chimera [software quality]," *IEEE Software*, vol. 13, no. 1, pp. 33–43, 1996.

[22] T. Haigh and C. Landwehr, "Building code for medical device software security," *IEEE Cybersecurity*, 2015.

[23] C. E. Landwehr and A. Valdes, "Building code for power system software security," *Technical Report. IEEE Computer Society*, 2017.

[24] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer, "What good are strong specifications?" in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 262–271. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606572

[25] R. Mohanani, P. Ralph, and B. Shreeve, "Requirements fixation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 895–906. [Online]. Available: https://doi.org/10.1145/2568225.2568235

[26] Y. Acar, C. Stransky, D. Wermke, M. L. Mazurek, and S. Fahl, "Security developer studies with GitHub users: Exploring a convenience sample," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, 2017, pp. 81–95.

[27] S. Oesch and S. Ruoti, "That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers," in *Proc. of USENIX Security Symp*, 2020.

[28] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor, "Encountering stronger password requirements: user attitudes and behaviors," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, 2010, pp. 1–20.

[29] B. Ur, G. Kelley, S. Komanduri, J. Lee, M. Maase, M. L.Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F.Cranor, S. Egleman, and J. López, "Helping users create better passwords," *USENIX ;login:*, 2012.

[30] B. Ur, F. Noma, J. Bees, S. M. Segreti, R. Shay, L. Bauer, N. Christin, and L. F. Cranor, "" i added'!'at the end to make it secure": Observing password creation in the lab," in *Eleventh Symposium On Usable Privacy and Security ({SOUPS} 2015)*, 2015, pp. 123–140.

[31] B. Ur, J. Bees, S. M. Segreti, L. Bauer, N. Christin, and L. F. Cranor, "Do users' perceptions of password security match reality?" in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 3748–3760.

[32] B. Ur, F. Alfieri, M. Aung, L. Bauer, N. Christin, J. Colnago, L. F. Cranor, H. Dixon, P. Emami Naeini, H. Habib *et al.*, "Design and evaluation of a data-driven password meter," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017, pp. 3775–3786.

[33] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 538–552.

[34] S. Egelman, A. Sotirakopoulos, I. Muslukhov, K. Beznosov, and C. Herley, "Does my password go up to eleven? the impact of password meters on password selection," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013, pp. 2379–2388.

[35] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman, "Of passwords and people: measuring the effect of password-composition policies," in *Proceedings of the sigchi conference on human factors in computing systems*, 2011, pp. 2595–2604.

[36] M. Weir, S. Aggarwal, M. Collins, and H. Stern, "Testing metrics for password creation policies by attacking large sets of revealed passwords," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 162–175.

[37] C. Weir, B. Hermann, and S. Fahl, "From needs to actions to secure apps? the effect of requirements and developer practices on app security," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[38] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on Android application security," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 121–136.

[39] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do Java developers struggle with cryptography APIs?" in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 935–946.

[40] 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 & communications security*. ACM, 2013, pp. 73–84.

[41] N. Patnaik, J. Hallett, and A. Rashid, "Usability smells: An analysis of developers' struggle with crypto libraries," in *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[42] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[43] E. E. Cureton, "Rank-biserial correlation," *Psychometrika*, vol. 21, no. 3, pp. 287–290, 1956.

[44] A. L. Strauss and J. M. Corbin, *Basic of qualitative research: Techniques and procedures for developing Grounded Theory*. Sage Publications, 1998.

[45] B. G. Glaser and A. L. Strauss, *The discovery of Grounded Theory: strategies for qualitative research*. New York: Aldine de Gruyter, 1967.

[46] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez, "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 523–537.

[47] A. Naiakshina, A. Danilova, E. Gerlitz, and M. Smith, "On conducting security developer studies with CS students: Examining a password-storage study with CS students, freelancers, and company developers," in *CHI Conference on Human Factors in Computing Systems*, R. Bernhaupt, F. F. Mueller, D. Verweij, J. Andres, J. McGrenere, A. Cockburn, I. Avellino, A. Goguey, P. Bjøn, S. Zhao, B. P. Samson, and R. Kocielnik, Eds. Honolulu, HI: ACM, 2020, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3313831.3376791

[48] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[49] C. Spearman, "The proof and measurement of association between two things," *Studies in individual differences: The search for intelligence*, 1961.

[50] M. Hazhirpasand, M. Ghafari, S. Krüger, E. Bodden, and O. Nierstrasz, "The impact of developer experience in using Java cryptography," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–6.

[51] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[52] J. Cox, "Password storage methods," 2017. [Online]. Available: https://medium.com/@jcox250/password-storage-d480309ca08f

[53] C. Wijayarathna and N. A. G. Arachchilage, "Why johnny can't store passwords securely? a usability evaluation of bouncycastle password hashing," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, ser. EASE'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 205–210. [Online]. Available: https://doi.org/10.1145/3210459.3210483

[54] 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.

[55] A. Adams and M. A. Sasse, "Users are not the enemy," *Communications of the ACM*, vol. 42, no. 12, pp. 40–46, 1999.

[56] Google, "Tink." [Online]. Available: https://github.com/google/tink

[57] S. Schmieg, "This issue demonstrates nicely how software engineers and cryptographers have a completely different idea about what a hash function does. for many software engineers, a hash function is a "one-way" function, with the output being essentially meaningless." August 2020, tweet. @SchmiegSophie. [Online]. Available: https://mobile.twitter.com/SchmiegSophie/status/1292930642561265664

[58] K. Mindermann and S. Wagner, "Fluid intelligence doesn't matter! Effects of code examples on the usability of crypto APIs," in *42nd International Conference on Software Engineering (ICSE) Posters*, 2020, poster.

[59] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, "OAuth 2.0 security best current practice," IETF Web Authorization Protocol, Tech. Rep. draft-ietf-oauth-security-topics-16, 2020. [Online]. Available: https://www.ietf.org/id/draft-ietf-oauth-security-topics-16.html

[60] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 378–390.

[61] S. S. Chow, Y.-J. He, L. C. Hui, and S. M. Yiu, "SPICE–simple privacy-preserving identity-management for cloud environment," in *International Conference on Applied Cryptography and Network Security*. Springer, 2012, pp. 526–543.

[62] M. Isaakidis, H. Halpin, and G. Danezis, "UnlimitID: Privacy-preserving federated identity management using algebraic MACs," in *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, 2016, pp. 139–142.

[63] C. Landwehr, "We need a building code for building code," vol. 58, no. 2, 2015. [Online]. Available: https://doi.org/10.1145/2700341