SLR: From Saltzer and Schroeder to 2021…47 Years of Research on the Development and Validation of Security API Recommendations

NIKHIL PATNAIK, University of Bristol, UK
ANDREW DWYER, University of Durham, UK
JOSEPH HALLETT and AWAIS RASHID, University of Bristol, UK

Producing secure software is challenging. The poor usability of security Application Programming Interfaces (APIs) makes this even harder. Many recommendations have been proposed to support developers by improving the usability of cryptography libraries—rooted in wider *best practice* guidance in software engineering and API design. In this SLR, we systematize knowledge regarding these recommendations. We identify and analyze 65 papers, offering 883 recommendations. Through thematic analysis, we identify seven core ways to improve usability of APIs. Most of the recommendations focus on helping API developers to *construct* and *structure* their code and make it more usable and easier for programmers to *understand*. There is less focus, however, on *documentation*, *writing requirements*, *code quality assessment*, and the impact of *organizational software development practices*. By tracing and analyzing paper ancestry, we map how this knowledge becomes validated and translated over time. We find that very few API usability recommendations are empirically validated, and that recommendations specific to usable security APIs lag even further behind.

CCS Concepts: • **Security and privacy**;

Additional Key Words and Phrases: API, usability, security, SLR, recommendations

ACM Reference format:
Nikhil Patnaik, Andrew Dwyer, Joseph Hallett, and Awais Rashid. 2023. SLR: From Saltzer and Schroeder to 2021…47 Years of Research on the Development and Validation of Security API Recommendations. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 60 (April 2023), 31 pages.
https://doi.org/10.1145/3561383

1 INTRODUCTION

Programming is hard to do well, and, even more so, securely. Developers frequently combine functions from **Application Programming Interfaces (APIs)**, but some are notoriously difficult to use correctly [50, 73], with cryptography and security libraries often singled out as being particularly obtuse [44, 59]. Strategies ranging from Gamma et al.'s design patterns [25] to the design

Authors' addresses: N. Patnaik (corresponding author), J. Hallett, and A. Rashid, University of Bristol, UK; emails: nikhil.patnaik@bristol.ac.uk, joseph.hallett@bristol.ac.uk, awais.rashid@bristol.ac.uk; A. Dwyer, University of Durham, UK; email: andrew.dwyer@durham.ac.uk.

1049-331X/2023/04-ART60 $15.00
https://doi.org/10.1145/3561383

principles of Saltzer and Schroeder [76] have been influential on software engineering practices as well as security API design. We investigate how such strategies may have informed recommendations for designing APIs, especially those APIs that provide security and cryptographic functionality.

Over the past 10 years, to help API designers produce security APIs that are more usable, various papers have proposed *usability guidelines, principles and recommendations* [31, 53, 56, 70]—hereafter *recommendations*.

Tracing the ancestry of papers offers a means to systematize knowledge that inform recommendations for improving usability of security APIs: providing a deeper understanding of current areas of focus, how these have been validated, and where more evidence may be required.

Although previous studies have highlighted existing guidance available to developers [85], no work, to date, has systematized knowledge across this guidance, traced ancestral relationships, as well as the impact of such ancestry on current recommendations for security API usability.

Our SLR begins with 13 papers that provide *Security API designer recommendations*. We trace and analyze their ancestry, identifying 883 recommendations in 65 papers (categorized in Table 1). These include papers offering general API design recommendations, those providing general security best practice, and broader software engineering design guidance and recommendations. We categorize these recommendations and analyze their ancestry to answer three research questions:

RQ1: What do current recommendations focus on? Using thematic analysis, we develop 36 descriptive themes across 883 recommendations. These 36 themes are consolidated into seven broad categories. While many papers recommend improving documentation to assist developers [7, 10, 11, 56, 60, 69, 70, 73, 86, 89], we see how this is reflected in *Security API designer* papers. For instance, we find that only 17% of Security API designer recommendations focus on aspects related to code's *documentation*, whereas 36% of these recommendations address the code's *construction*.

RQ2: How, and to what extent, have various recommendations been validated? Reviewing recommendations by different paper types shows that less than a quarter (across the whole corpus) have been empirically validated. Empirical validation is the scientific method of verifying a theory through thorough application to test its effectiveness and validity. Only three papers from *Security API designer* guidance fall within this category. We also identify areas of guidance that seem to receive greater focus from the research community regarding empirical validation and where potential gaps may lie.

RQ3: What are the implications of this coverage, in terms of ancestry and their validation, for the emerging set of Security API designer recommendations? In tracing how the guidance has developed over time, we find extended *ancestry chains*—histories where literature has built on prior work—for almost half of these papers, however, empirical validation is limited within those chains. We explore how these ancestries develop—by addressing usability challenges arising from APIs relating to particular languages, specific security problems pertaining to particular applications, or via experiences from developing security analysis and verification tools.

Our key findings show:

- A breakdown of the different guidance for improving the usability of security APIs (Table 4). We show that API and software engineering-focused papers tend to focus on how one structures code and makes it understandable, only the security-focused papers consider organizational factors, and that security-focused papers specifically touching on APIs tend to ignore documentation or validation aspects (RQ1: Section 4.1).
- A lack of empirical validation across the field. Empirical validation tests recommendations and prevents ineffective recommendations from being propagated over time. However, only 22% of the guidance is empirically validated in the literature through independent experimentation (RQ2: Section 4.2).

Table 1. Count of Recommendations (Papers) Analyzed in the Study, Broken Down by Category of Paper

Paper Category	Description	Count	Papers
Security API designer recommendations	Literature that explicitly provides recommendations for improving the usability of security APIs through design.	84 (13)	[1, 13, 22, 26, 31, 37, 53, 54, 56, 64, 66, 70, 87]
API designer recommendations	Literature that focuses on APIs, which may include limited elements of general practice that permits *good* security, but does not explicitly attend to security itself.	285 (15)	[7, 10, 11, 17, 20, 34, 40, 47, 50, 63, 69, 73, 74, 85, 89]
Software engineering recommendations	Literature that is around generic software engineering and best practices in the form of recommendations.	207 (17)	[3, 16, 19, 24, 25, 32, 33, 39, 41, 42, 45, 57, 58, 60, 61, 71, 83]
Security engineering recommendations	Literature in software engineering and computer security that explicitly make related recommendations but does not specifically focus on API security.	307 (20)	[4, 6, 12, 15, 23, 27, 29, 36, 38, 49, 51, 52, 55, 67, 68, 75, 76, 79, 80, 86]
Total		883 (65)	

- A well defined ancestry linking much of the security API guidance going back 47 years to Saltzer and Schroeder's seminal work in 1974 [75], amongst others (RQ3: Section 4.3).

Through a systematic literature review, we identify and analyze the challenges addressed by the 883 recommendations and trace the origins of the recommendations from our 13 security API designer papers. Our SLR results in a set of *eight meta-recommendations* that summarize 47 years of design guidance targeted at software engineering and computer security.

Forty-seven years later, we find that much of the current guidance is still dealing with issues Saltzer and Schroeder and other earlier works raised almost half a decade ago.

2 BACKGROUND: AN INTRODUCTION TO SALTZER AND SCHROEDER'S PRINCIPLES

Our SLR begins with 13 papers that present recommendations to help improve the design of *Security APIs*. But how did these recommendations come to be? As we trace the ancestry of these recommendations, we discovered a number of papers from which the ancestries stemmed, the earliest work being that of Saltzer and Schroeder [75].

2.1 Saltzer and Schroeder's Design Principles

In 1974, Saltzer addressed the challenge of protection and control of information sharing in Multics (Multiplexed Information and Computing Service). Saltzer offered five design principles to help evaluate different designs. These design principles addressed access control lists, identification and authentication of users, hierarchical control of access specifications, and primary memory protection. In 1975, Saltzer and Schroeder presented eight design principles and a series of desired functions with the intention of protecting computer-stored information from unauthorised access and modification. At the time software application could store information and be simultaneously

Table 2. Saltzer and Schroeder's Eight Principles to Guide the Design of Information Protection Mechanisms in Computer Systems [76]

Principle	Description
Economy of Mechanism	Keep the design as simple as possible. Design and implementation errors that result in unwanted access paths will not be noticed during normal use (since normal use usually does not include attempts to exercise improper access paths).
Fail-safe defaults	Base access decisions on permission rather than exclusion.
Complete mediation	Every access to every object must be checked for authority.
Open design	The design should not be secret. The mechanism should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords.
Separation of privilege	Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.
Least privilege	Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error.
Least common mechanism	Minimize the amount of mechanism common to more than one user and depended on by all users. Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security.
Psychological acceptability	It is essential that the interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

used by several users. The key challenge Saltzer and Schroeder wanted to address was that of multiple use. For applications with users who do not have equal authority, a system is needed to enforce the desired authority structure in the application [76]. Saltzer and Schroeder's work became very influential and applicable to a wide range of fields such as enforcing security policies [78], evaluating the Java security architecture [28], and minimising user-related faults in information systems security [82]. However, it was in 1995 when Saltzer and Schroeder's principles were first applied to the design of security APIs through Cryptlib [35]. In 1995, Gutmann designed Cryptlib, a cryptographic library, based on the adaptation of Saltzer and Schroeder's principles [35].

3 METHOD

3.1 Identifying the 13 Security API Designer Papers

3.1.1 Step 1: Online Search. We used Google Scholar, IEEExplorer, and ACM Digital Library with the following search terms to select papers that offer Security API designer recommendations:

$$\texttt{API}\ \left(\substack{\texttt{Usability|Guidelines|}\\ \texttt{Principles|Design|Librar(y|ies)}}\right)\texttt{? (Security)?}$$

We used a snowball-method [88] on the papers identified from the online search; and using forward and backward snowballing, we checked to see if there were other relevant papers.

3.1.2 Step 2: Review of relevant journals and conferences. To ensure that the online search did not miss any key works, we reviewed ten relevant venues from their first issue to their latest available (November 2021) and added any paper that appeared to offer Security API designer recommendations to our initial set. We also ran a snowball method [88] on the papers we found through our review of these 10 venues to find more relevant papers. We reviewed the following venues as they represented the leading security and software engineering venues from the ACM, IEEE, and USENIX communities:

- ACM **Transactions on Software Engineering and Methodology (TOSEM)**,
- IEEE Symposium on Security and Privacy (Oakland),
- IEEE **Transactions on Software Engineering (TSE)**,
- **International Conference on Software Engineering (ICSE)**,
- **USENIX Security Symposium (USENIX)**,
- International **Symposium on Usable Privacy and Security (SOUPS)**, and
- ACM Conference on **Computer and Communications Security (CCS)**,
- **Network and Distributed System Security Symposium (NDSS)**,
- ACM **Foundations of Software Engineering (FSE)**,
- ACM Conference on **Human Factors in Computing Systems (CHI)**

We reviewed these ten venues as an additional check to build on top of the initial set of papers identified through the Google Scholar/IEEEXplorer/ACM DL search. Papers from other venues were identified through the snowballing process (Table 3). We identified an initial 45 candidate papers, through step 1 and 2, that provided Security API Designer recommendations.

3.1.3 Step 3: Selecting Relevant Work. Each paper was reviewed independently by three authors. During the review, we used the following inclusion and exclusion criteria to decide whether a paper would be included in our list of Security API designer papers or not.

Inclusion:

- The paper gave recommendations about improving an API or programming interface.
- The recommendations aimed to improve the usability of the security API.
- The API was designed to be used by programmers for building an application, rather than end-users using a program for security tasks (e.g., to accomplish PGP encryption).

Exclusion:

- The recommendations were not about APIs, but rather a technology an API might wrap (e.g., the use of various cryptographic modes such as ECB).
- The recommendations were targeted at improving the engineering quality of an API rather than security directly. Whilst a secure API is often a well engineered API, the recommendations did not focus on security but rather broader engineering concerns (e.g., reducing an API's size to a few clear methods may reduce confusion, and be easier to verify—but unless the paper explicitly stated that this was for security, it would not be included).
- The recommendations must discuss an API—several papers gave security recommendations for configuration management that were similar to recommendations for securing APIs; but these papers focused on advising IT workers who maintain these systems and did not describe *programming* interfaces.

Table 3. Where Each of the 65 Papers Providing the 883 Actionable Recommendations for Usability and Security Are Published

Venue or Publisher	Count	References
IEEE S&P	6	[1, 4, 13, 31, 52, 53]
USENIX Security	5	[36, 37, 39, 66, 87]
IEEE Software	4	[50, 61, 73, 86]
SOUPS	4	[6, 29, 64, 70]
IEEE VL/HCC	3	[7, 47, 85]
CACM	3	[57, 58, 75]
ACM CCS	3	[22, 23, 26]
IEEE TSE	2	[38, 41]
Carnegie-Mellon University	2	[69, 79]
Elsevier VL&C	2	[3, 33]
OWASP	2	[67, 68]
Addison Wesley	2	[10, 24]
IEEE/ACM ICSE	1	[54]
ACM OOPSLA	1	[11]
IEEE QRS	1	[56]
ECOOP	1	[25]
ACM CHI	1	[60]
PPIG	1	[17]
People and Computing	1	[32]
ACM SIGDOC	1	[45]
EclipseCon	1	[20]
ACM CCSC	1	[63]
HCSE	1	[34]
IJCSNS	1	[89]
Microsoft	1	[55]
BSIMM	1	[15]
Proceedings of the IEEE	1	[76]
ACM Software Engineering Notes	1	[51]
IEEE SESS	1	[12]
Secure Software, Inc.	1	[80]
IEEE ACSAC	1	[49]
IEEE COMPSAC	1	[42]
Human-Computer Interaction	1	[71]
ACM TOIS	1	[16]
National Computer Conference	1	[83]
Pearson Education	1	[19]
ACM Queue	1	[40]
ESE	1	[74]
IEEE SCAM	1	[27]
Overall	65	

- The recommendations given were too generic to offer any meaningful advice (e.g., “an API must be secure”—we agree, but this recommendation offers no advice on how to achieve this).

Although our 13 Security API designer papers meet our inclusion and exclusion criteria, the context of each paper and the challenges they address range greatly. For example, Brown et al. work is categorised as a Security API designer paper [13]. Brown et al. identified a series of effective checks to find bugs in **JavaScript (JS)** run-time systems like; Node.js, Blink, and PDFium. Brown et al. also developed a library with a usable security API, that prevented bugs without imposing significant overhead. Brown et al. further goes on to explain that the bugs are not explicit to JS, but instead said that identical challenges and flaws could be found in other scripting languages like Ruby and Python, concluding that a more principled API design would benefit those languages as well.

Georgiev et al. identified poorly designed SSL implementations vulnerable to MITM attacks [26]. Georgiev et al. analysed applications that consisted of general APIs interacting with security APIs, like the Crypto API and SSL APIs. Georgiev et al. concluded that the vulnerabilities are due to poorly designed security APIs, and provided recommendations to address Security API design. Through the inclusion and exclusion criteria, 13 Security API designer papers were taken forward.

3.2 Tracing the Ancestry of the 13 Security API Designer Papers

We identified earlier works that influenced the recommendations of the 13 Security API designer papers by following the cited works associated with the recommendations. We repeated this process on the recommendations of these earlier works and continued until no more recommendations were offered and we reached the origin of the recommendations from the 13 Security API designer papers. For every paper we identified, along the ancestry chain, we checked if these works were validated or referred to by other works (Table 6). At the end of this step, we had 156 papers including the 13 Security API designer papers (Table 1, Figure 1).

3.3 Identifying Actionable Recommendations

We sought papers that provide specific steps to improve APIs, rather than general engineering guidance. From the 156 papers offering recommendations through the initial search and snowballing, we narrowed our recommendations to those that are *actionable*—that is they detail specific and clear steps to improving the usability of an API, such as:

Improved Usability.

> "Easy to use, even without documentation: Developers like end-users do not like to read manuals [· · ·]. If the API is not self-explanatory or worse gives the false impression that it is, developers will make dangerous mistakes." [31]

Offered Guidance.

> "Economy of mechanism: Keep the design as simple and small as possible. [· · ·] design and implementation errors that result in unwanted access paths will not be noticed during normal use [· · ·]. As a result, techniques such as line-by-line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential." [76]

Recommendations that were too *general* (i.e., not actionable), such as a general principle that should be taken into account to improve the usability of an API without detailing specifics about how that principle should be implemented were excluded, such as:

Developing General Principles.

> "If it's hard to find good names, go back to the drawing board." [11]

From the 156 papers, we identified 65 papers providing 883 actionable recommendations for improving usability and security (Table 3).

From our 65 actionable papers, two authors allocated each paper to one of four paper types, shown in Table 1, based on an inductive process derived from the papers themselves. This results in *Security API designer* and three additional paper types. This process offers a high-level overview of what each paper category broadly addresses and helps us to understand how recommendations propagate against different communities.

3.4 Categorizing the Recommendations

To understand the different areas of recommendation focus, we categorized our 65 actionable papers. To alleviate bias from predefined categorization the analysis followed an inductive, bottom-up approach [72], drawing out recommendation themes.

(1) Two authors, in joint discussion, selected 50 recommendations to identify different *codes* to build a mutual understanding of the recommendations. An initial *codebook* [2] with 28 codes was created.
(2) Over three iterations, the 883 recommendations were categorized using the initial codebook. Additional codes were created to capture the diversity of recommendations identified during the process. The initial codebook was updated to include 19 categories and 75 descriptive sub-categories.
(3) Through a visual whiteboard mapping discussion between three authors, the number of codes were reduced and amalgamated. This resulted in a consolidated codebook with seven categories, and 36 descriptive sub-categories, as shown in Table 4. The mappings of categories onto recommendations was updated using the new codebook.

Most recommendations were assigned a single top-level category and one of its sub-categories. A sixth ($\frac{162}{883}$, 18%) were more complex—exhibiting multiple elements for API design, security or general software engineering guidance into one—and so two categories were used. No recommendation required more than two top-level categories. For example, Pane and Myers [69] recommend supporting novice programmers:

> "Use Signaling to Highlight Important Information."

This was assigned to the *Understanding: Drawing Attention* category and descriptor sub-category as it is concerned with helping the developer identify relevant information. Later, Pane and Myers also recommend:

> "Help detect, diagnose, and recover from errors."

This was assigned two categories: *Understanding: Assist* as the recommendation concerns developer assistance to diagnose problems, and *Construction: Error Handling* as it deals with recovery from errors.

To validate our categories, a random 10% sample of the recommendations were assessed by an independent coder. Using *Cohen's* κ (a common measure of interrater reliability [18]), and using only a single category per recommendation (as Cohen's κ only deals with single categorizations per subject), we calculated a κ of 0.74 when mapping the categories, and 0.76 when mapping the descriptors—indicating substantial agreement [48].

4 FINDINGS

Table 4 outlines the seven recommendation categories and 36 descriptor sub-categories. The seven categories describe overarching themes and topics about which papers make recommendations.

Table 4. Codebook Showing **Categories** and • Descriptors for the Recommendations Identified

Code	Description
Assessment	**The quality and testing of software and APIs.**
• Quality Engineering	The development, good practice, and management of software and APIs.
• Quality Assurance	The methods and tools used to assess and audit software and APIs.
Construction	**The technical features of software and APIs.**
• Abstraction	Expressing different components of software and APIs through abstraction.
• Access Validation	Complete Mediation—Checking for access.
• Code	Any code or data involved in the construction of software and APIs.
• Error Handling	How software and APIs deal with errors.
• Economy of Mechanism	Ensuring minimalist and simple design of software and APIs.
• Open Design	Ensuring that it is clear what the design is.
• Technical Specifics	Any element not covered by the other *Construction* descriptors.
• Durability	How software and APIs develop over time, are maintained, and can be deprecated.
Default Secure	**The different methods and practices to develop security as a fundamental outcome.**
• Bug and Defect Management	Processes and practices for the handling of bugs and defects.
• Fail-Safe Default	How does software or an API ensure that it will always provide, by default, the safest option?
• Secure Architecture	How software and API architecture is designed with security at its core.
• Compartmentalization	Least Common Mechanism—Ensuring that things are not unnecessarily shared.
Documentation	**Documentation methods and practices.**
• Explain	How well documentation describes the usage of an API or software.
• Inventory / COTS	The development of an inventory to record the different components of an API and software.
• Telemetry and Reporting	Ensuring active collection and recording of data and information.
• Publish and Communicate	The use of documentation to distribute or to offer information.
• Standardized	Ensuring that documentation provides cohesive standards.
• Exemplars	The use of examples (frequently code) to help explain different aspects of software and APIs.
• Guidance	The development of guidance for users or designers.
Organizational Factors	**How organizations respond to developing software and APIs and interface with external factors.**
• Incident Response	The development of practices to respond to emergencies or incidents from software and APIs.
• Security Practice	How an organization develops knowledge and practice of security.
• Training	The delivering of training for organizations and their members.
• Third Party	How organizations interface with third parties and third party components.
• Regulatory	Any regulatory, legal, or compliance that an organization does.
• Risk Assessment and Metrics	Assessing risk and developing metrics to measure it.
Requirements	**The development of requirements for software and APIs.**
• Implement Requirements	The implementation and application of requirements.
• Write Requirements	The construction, identification, and development of requirements.
Understanding	**How software and APIs come to be understood and practiced by humans.**
• Assist	Psychological Acceptability—how an API user or API developer deals with the load of programming and techniques to assist developers.
• Drawing Attention	Highlighting or pointing toward information required for proper or secure use of software and APIs.
• Misuse	The prevention of an API user or API developer misusing software and APIs.
• Relevant Information	The provision of information that concerns a particular task or object of study.
• Meaningful Options	The provision of options that make sense to API users.
• Sufficient Information	The provision of enough information to effectively communicate and provide understanding.
• Validation of Activity	Providing API users and API developers tools that check their activities.

The descriptor sub-categories offer greater detail within each of the categories. For example, the *Construction* category captures recommendations to help structure and build software. Its *Code* descriptor sub-category focuses on particular programming details. Bloch, for instance, advises developers to:

> "Return zero-length arrays, not nulls." [10]

In contrast, the *Economy of Mechanism* descriptor identifies the simple code construction to avoid errors—found in Nino et al.'s *"be minimal"* [63], Grill et al.'s *"Do not provide multiple ways to achieve one thing"* [34], OWASP's *"Keep It Simple, Stupid Principle"* [67] or Saltzer and Schroeder's *Economy of Mechanism* principle [76]; from which we name the descriptor. We also observe recommendations on how to document code (the *Documentation* category)—typically focused on clear explanation, communication, standardization and exemplars. Recommendations also assist API users' *Understanding* by aligning concepts with their mental models and helping them with the cognitive load of programming (the *Assist* descriptor); for example, Ko et al. recommend:

> "Help programmers recover from interruptions or delays by reminding them of their previous actions." [3]

Other topics in the *Understanding* category include *Drawing Attention* to *Relevant* and *Sufficient* information, as well as providing *Meaningful Options*.

Table 5 shows the number of recommendations in each paper type mapped to each category and sub-category. We find that, over the 883 recommendations, the majority concern the construction and structure of code (the *Construction* category, 32%), as well as helping to make the code easier to comprehend and clear to the developer (the *Understanding* category, 23%). The remaining recommendations are more or less evenly spread across the five remaining categories (ranging between 7% and 14%).

4.1 RQ1: What Do Current Recommendations Focus on?

Take-away 1:

- API designer and Software engineering papers focus on how to *structure* code and how to make it *understandable*.
- Practically only Security engineering papers make recommendations about *Organizational Factors*.
- Security API designer papers do not engage sufficiently with various aspects of *Documentation* or *Understanding: Validation of Activity*, which should be addressed in future work.

Recommendations, as derived from our literature search and ancestral tracing, tend to favor technical aspects. We break these down by paper type to see the areas they focus on and how Security API designer literature compares to other communities. Both *API designer* and *Security API designer* paper types offer more recommendations on API *Construction* and its *Code*. The *Construction* category is associated with 57% of API designer and 36% of Security API designer paper types, with *Understanding* covering 21% and 24% of the paper types, respectively. As API-related recommendations are likely to deal with the interface with code, it is reasonable to expect these paper types to focus more on *Construction*. Recent work on recommendations for security API usability [31, 70] suggest that we may be witnessing a move to recommendations around improving code usability (*Understanding*), but this is limited by the number of papers in this type (13—see Table 1).

For recommendations in *Software engineering guidance*, this relationship is reversed. A greater emphasis is placed on *Understanding* (54%), with a reduction in a focus on *Construction* (25%).

Table 5. Recommendations Mapped to Category and Paper Type

Recommendation Category	Security API designer	API designer	Software engineering	Security engineering	Overall
Total	84	285	207	307	883
Assessment	8 (10%)	20 (7%)	18 (9%)	76 (25%)	122 (14%)
• Quality Engineering	3 (4%)	3 (1%)	4 (2%)	36 (12%)	46 (5%)
• Quality Assurance	5 (6%)	17 (6%)	14 (7%)	40 (13%)	76 (9%)
Construction	30 (36%)	163 (57%)	52 (25%)	35 (11%)	280 (32%)
• Abstraction	1 (1%)	2 (1%)	3 (2%)	0	6 (1%)
• Access Validation	5 (6%)	2 (1%)	1 (1%)	10 (3%)	18 (2%)
• Code	14 (17%)	79 (28%)	11 (5%)	8 (3%)	112 (13%)
• Error Handling	1 (1%)	11 (4%)	1 (1%)	0	13 (2%)
• Economy of Mechanism	5 (6%)	18 (6%)	32 (16%)	5 (2%)	60 (7%)
• Open Design	1 (1%)	1 (<1%)	0	4 (1%)	6 (1%)
• Durability	2 (2%)	18 (6%)	3 (2%)	5 (2%)	28 (3%)
• Other	1 (1%)	32 (11%)	1 (1%)	3 (1%)	37 (4%)
Default Secure	9 (11%)	5 (2%)	8 (4%)	42 (14%)	64 (7%)
• Bug and Defect Management	1 (1%)	0	2 (1%)	6 (2%)	9 (1%)
• Fail-Safe Default	4 (5%)	0	1 (1%)	4 (1%)	9 (1%)
• Secure Architecture	4 (5%)	3 (1%)	5 (2%)	28 (9%)	40 (5%)
• Compartmentalization	0	2 (1%)	0	4 (1%)	6 (1%)
Documentation	14 (17%)	28 (10%)	11 (5%)	40 (13%)	93 (11%)
• Explain	3 (4%)	10 (4%)	8 (4%)	2 (1%)	23 (3%)
• Inventory / COTS	0	2 (1%)	0	4 (1%)	6 (1%)
• Telemetry and Reporting	0	0	0	13 (4%)	13 (2%)
• Publish and Communicate	1 (1%)	1 (<1%)	0	9 (3%)	11 (1%)
• Standardized	1 (1%)	3 (1%)	0	5 (2%)	9 (1%)
• Exemplars	3 (4%)	6 (2%)	0	0	9 (1%)
• Guidance	6 (7%)	6 (2%)	3 (2%)	7 (2%)	22 (3%)
Organizational Factors	3 (4%)	0	0	51 (17%)	54 (6%)
• Incident Response	0	0	0	5 (2%)	5 (1%)
• Security Practice	1 (1%)	0	0	12 (4%)	13 (2%)
• Training	2 (2%)	0	0	13 (4%)	15 (2%)
• Third Party	0	0	0	1 (<1%)	1 (<1%)
• Regulatory	0	0	0	7 (2%)	7 (1%)
• Risk Assessment and Metrics	0	0	0	13 (4%)	13 (2%)
Requirements	0	10 (4%)	6 (3%)	55 (18%)	71 (8%)
• Implement Requirements	0	4 (1%)	0	3 (1%)	7 (1%)
• Write Requirements	0	6 (2%)	6 (3%)	52 (17%)	64 (7%)
Understanding	20 (24%)	59 (21%)	112 (54%)	8 (3%)	199 (23%)
• Assist	6 (7%)	36 (13%)	52 (25%)	2 (1%)	96 (11%)
• Drawing Attention	2 (2%)	1 (<1%)	8 (4%)	0	11 (1%)
• Misuse	2 (2%)	5 (2%)	5 (2%)	0	12 (1%)
• Relevant Information	1 (1%)	2 (1%)	11 (5%)	1 (<1%)	15 (2%)
• Meaningful Options	4 (5%)	11 (4%)	27 (13%)	1 (<1%)	43 (5%)
• Sufficient Information	0	1 (<1%)	3 (2%)	1 (<1%)	5 (1%)
• Validation of Activity	5 (6%)	3 (1%)	6 (3%)	3 (1%)	17 (2%)

Some recommendations were assigned multiple categories. All of the recommendations were assigned to at least one category. Each percentage represents the percent of recommendations of a given paper type that are in a given category.

Unlike other paper types, *Security engineering guidance* focused less on *Construction* and *Understanding* (11% and 3%, respectively), and instead offered greater attention to other categories such as *Assessment* (25%) and *Requirements* (18%).

Recommendations categorized under *Organizational Factors* are almost exclusively derived from Security engineering papers. These recommendations deal with processes such as Incident Response, Developer Training, and Risk Assessment. Many of the Security Engineering papers are corporate literature presented by the security engineering teams at organizations such as Microsoft [55]. These recommendations are derived from day-to-day experience with practical tasks such as training developers and writing policy. The ancestry of the recommendations presented by many of the corporate literature is almost non-existent, because they are based on experience.

The relationship between corporate literature and academic literature is present in Security engineering papers, in Tondel [86] and Assal [6] (Figure 1). Many of the recommendations in this category come from corporate grey literature (such as Microsoft's SDL [55], The BSIMM framework [15] and OWASP [68]). For example, Microsoft's SDL encourages developers to *"Establish a standard incident response process"* [55] so that there are mechanisms for dealing with software defects when they are inevitably discovered (which we capture under the *Incident Response* descriptor). BSIMM recommends that organizations should *"educate executives"* [15] so that *decision-makers* in an organization are sufficiently knowledgeable about security (*Organizational Factors: Training*). Recommendations focused on organizations were almost exclusively limited to Security engineering papers; suggesting greater emphasis in the security community on considering the wider implications for developers and software in organizations and the impact of external contexts on being secure. The recommendations reflect how organizational factors may affect the design of security APIs.

When looking at the areas Security API designer papers focus on, we find that, in the *Construction* category, there is a greater emphasis on the *Code* sub-category. This shows that the research community are strongly focusing on what challenges developers face when working with Security APIs, and on the writing of code and implementing the functions of the security APIs. The applications of this challenge are studied in more depth by Georgiev et al. who provide recommendations to mitigate and resolve the issue for various stakeholders [26]. Georgiev et al. identified that the SSL certificate validation protocol was broken in many security-critical applications available to the public. Software and applications such as Amazon's EC2 Java library, including all cloud clients, Paypal's SDK's that processed financial information, and many more Android applications and libraries were vulnerable to Man-in-the-Middle attacks. Georgiev et al. confirmed all Man-in-the-Middle attack-based vulnerabilities empirically [26].

4.2 RQ2: Are We Validating Recommendations?

Take-away 2:

- Today's Security API designer recommendations build upon those presented in historical papers. However, across this ancestry, only 22% of the papers are empirically validated, meaning further work should be conducted to strengthen their foundations.
- Of the Security engineering papers that receive empirical validation or are part of an ancestor–descendant relationship, over half are through an adaptation of recommendations.
- Only 3 of the 13 Security API designer paper have been empirically validated.
- In the absence of empirical validation, the four ancestor-descendant relationships can risk propagating ineffective recommendations.

Table 6. Codebook Used for Describing Five Different Kinds of Ancestor–Descendant Relationships Between Papers, Including Empirical Validation

Code	Description
Distillation	Descendant condenses an Ancestor's recommendation to further build upon it by addressing a more specific challenge.
Borrowed	Descendant addresses Ancestor's guidelines at a superficial level either to review or to run an experiment and analyze their results.
Adaptation	Descendant has translated recommendations from an Ancestor, either through re-wording or forming their own recommendations directly derived from the Ancestor.
Comparison	Descendant contrasts Ancestor recommendation with another set of recommendations, perhaps written by another Descendant. Or Descendant compares their recommendation to that of their Ancestor.
Empirical	Descendant has experimented and evaluated through research an Ancestor recommendation. Descendant assesses whether the Ancestor's recommendations is valid.

To assess the relationships between different recommendations over time, we constructed their ancestry by separating each recommendation's inheritance into five distinct ancestor–descendant relationships (Table 6). Within the ancestor–descendant relationships, we make a distinction between empirical validation and the four remaining relationships. With empirical validation it is possible to test recommendation effectiveness through experimentation or systematic observation. However, this bar is lower for the other ancestor–descendant relationships. These are presented in Table 6. Below are examples of how each ancestor–descendant relationship and empirical validation relate to the literature.

Distillation. An evolution occurs from Bloch's 2001 book *Effective Java* [10] to his 2006 paper *How to Design a Good API and Why it Matters* [11]. The book on Java programming focuses on usage and is later condensed by the short paper that addresses the design of APIs through a series of API usability recommendations—with the latter paper used frequently by Security API designer papers.

Borrowed. Myers and Stylos [58] refer to the ancestry between Grill et al. [34] and Nielsen and Molich [62]:

> "Grill et al. described a method where they had experts use Nielsen's Heuristic Evaluation to identify problems with an API and observed developers learning to use the same API in the lab. An interesting finding was these two methods revealed mostly independent sets of problems with that API." [58]

Adaptation. In 2017 Acar et al. [1] evaluated and compared the usability of five Python-based cryptographic libraries. To evaluate the usability of these cryptographic libraries, Acar et al. *adapted* recommendations from Bloch [11], and from Green and Smith [31].

> "We adapt guidelines from these various sources [Bloch [11], Green and Smith [31]] to evaluate the APIs we examine." [1]

Table 7. Rates of Different Kinds of Paper Validation for Different Categories of Guideline Papers in the Literature

Validation	Security API designer guidance	API designer guidance	Software engineering guidance	Security engineering guidance	Overall
Distillation	8	7	8	1	24 (37%)
Borrowed	2	7	2	4	15 (23%)
Adaptation	2	9	7	16	34 (52%)
Comparison	7	1	3	4	15 (23%)
Empirical	3	1	8	2	14 (22%)
Overall	22 (33%)	25 (39%)	28 (43%)	27 (42%)	26 (40%)

The overall values account for single papers being validated multiple times.

Comparison. Smith [84] reflects on the work of Saltzer [75] along with the 1975 paper written with Schroeder [76] by *comparing* the *principles* of Saltzer and Schroeder to those of the then-contemporary recommendations in software security.

> "The following are new—or newly stated—principles compared to those described in 1975." [84]

Empirical. In 2019 Patnaik et al. [70] *empirically* evaluate the 10 principles designed to improve usability and security of APIs by Green and Smith [31].

> "An empirical validation of Green and Smith's principles showing when a principle is not being applied but also identifying issues that Green and Smith's principles currently do not capture." [70]

4.2.1 Validation by Paper Type. Table 7 summarizes the number of papers associated with empirical validation and the other ancestor–descendant relationships. The various relationships we identified through mappings are presented in Figure 1; these show the full ancestry of the recommendations.

Overall, 22% of the papers engage in empirical validation of prior work. 8 (53%) Software engineering and 2 (47%) Security engineering papers are empirically validated, whereas only 3 Security API designer and 1 API designer papers are empirically validated. Recommendations written more recently, as part of the software engineering and the computer security community, have developed upon some form of ancestor–descendant relationship or empirical validation of Software engineering and Security engineering papers like Nielsen's usability heuristics [60] and Saltzer and Schroeder's principles [76]. Though efforts have been made to empirically validate earlier papers [60, 76], contemporary API recommendations are inherited from a large corpus of papers, where only 22% are empirically validated. This raises the need to further understand and validate how API recommendations are built. Of the 13 Security API designer papers, only 3 [22, 26, 31] are empirically validated.

As we create further recommendations, we must consider the role of ancestry, and the validation of what it recommends, to strengthen the foundations of Security API designer recommendations. Otherwise, we have no way of establishing if particular recommendations—and efforts invested in following them—have a material impact on improving the usability of security APIs. We also risk propagating ineffective recommendations over time. To understand to what extent we are propagating these ineffective recommendations, we need to learn what these ineffective recommendations are. During this section, we have primarily studied the extent to which empirical validation has been performed across the literature. However, there are four ancestor-descendant relationship that are commonly seen throughout the ancestries. We encourage researchers to engage with the literature by adapting, borrowing, comparing, and distilling older recommendations, but if the recommendations are not empirically validated, the ancestor-descendant relationships may risk propagating these ineffective recommendations. So before, adapting, borrowing, comparing, or distilling older recommendations, researchers should consider performing an empirical study to test the effectiveness of the recommendations.

An argument can be made that if a set of recommendations presented by a paper is the result of a well-validated long ancestry with many ancestor-descendant relationships along the chain, then this set of recommendations should be effective. We argue that if the set of recommendations have been empirically validated then designers can be assured that these recommendations are effective. However, if the set of recommendations is related to through multiple ancestor-descendant relationships, one should consider performing an empirical study to ensure that these recommendations are effective and the efforts to implement them for other challenges will not go to waste.

4.2.2 Which Aspects Are We Validating? If certain academic literature is not conducting extensive and in-depth validation of all areas, then which aspects are we validating? Table 8 counts the different recommendation categories broken down by their ancestor–descendant relationship, including empirical validation. Only 20% of the total 883 recommendations are empirically validated, and the 179 empirically validated recommendations found are rooted in only 14 papers (22%) (Tables 7 and 8).

We empirically validate more on *Construction* (45%) followed by *Understanding* (27%). For other ancestral relationships, categories exhibit different rates, but overall these are at the levels we would expect given their relative frequency across different paper types (Table 8). The software engineering and security communities should focus on forming more empirical and comparison-based relationships, as opposed to borrowing and distillation, to best ensure the effectiveness of the recommendations with thorough, repeatable experimentation (see Table 8).

4.3 RQ3: Where Do Security API Designer Recommendations Come From?

Take-away 3:

- Almost half of the Security API designer papers have a well defined and long ancestry, dating back to 1974.
- A distinction between the capacity to validate *abstract* and *concrete* recommendations (derived from experiences with particular tools and applications) exists.
- Recommendations derive mainly from *standalone* ancestries, or are processed through Gutmann [37] or subsequently through Green and Smith [31].

Table 8. Counts of Recommendations That Have Been Empirically Validated or Part of Other Ancestor–Descendant Relationships Across the Seven Broad Category Types

Recommendation Category	Distillation	Borrowed	Adaptation	Comparison	Empirical	Overall
Total	283	148	722	91	171	1415
Construction	162 (57%)	49 (33%)	284 (40%)	23 (25%)	79 (46%)	597 (42%)
Documentation	25 (9%)	35 (24%)	80 (11%)	9 (10%)	18 (11%)	167 (12%)
Requirements	11 (4%)	8 (5%)	61 (8%)	1 (1%)	3 (2%)	84 (6%)
Understanding	53 (19%)	41 (28%)	106 (15%)	28 (31%)	45 (26%)	273 (19%)
Assessment	17 (26%)	8 (1%)	102 (14%)	24 (16%)	21 (13%)	172 (12%)
Default Secure	15 (5%)	7 (5%)	34 (5%)	5 (6%)	5 (3%)	66 (5%)
Organisational and Regulatory Factors	0	0	55 (8%)	1 (1%)	0	56 (4%)

In the development of Security API designer recommendations, there are two broad forms—abstract and concrete—that we identified in the ancestries we analyzed.

First, *abstract* recommendations such as by Green and Smith [31] apply to a number of tools, applications, and contexts. Second, there are *concrete* recommendations as identified by the ancestries of tools and applications [22, 53]. These tend to be more tightly focused to a particular tool or application—and therefore offer advice, that as one would expect, focuses more exclusively on *Construction* and *Requirements*. For example:

Abstract Recommendation. "Defaults should be safe and never ambiguous." [31]
Concrete Recommendation. "Client-side validation must be thoroughly tested for consistency with server-side validation logic. WARDroid can help in identifying potential inconsistencies." [53]

From the examples given above, Green and Smith provide a recommendation for designing security APIs. The recommendation can be applied to tools, API design, and general practice by developers who are integrating security with their applications. However, Mendoza et al. offer a concrete recommendation. The recommendation is a policy expressed through WarDroid [53] and addresses API *Construction*. Ancestries tell us a complex story between concrete recommendations that may be easier to validate, but often are *standalone*, and broader recommendations that require a wide array of studies (over time) for validation. Furthermore, to devise a method for validating broader recommendations, one may need to refer back to the ancestry of these recommendations to understand the reason for their transformation over time.

Concrete recommendations may be easier to validate due to their association with a specific tool. The tool is presented as a solution that informs the recommendations the study provided. These recommendations can be empirically validated by validating the use of the tool. Making it easier to see any direct effects and assess the impact of recommendations.

Abstract recommendations are intentionally broader so that they can be applied to fields of software design and security. Studies that propose recommendations based on the insights of earlier papers that present abstract recommendations should dedicate their efforts to validating the recommendations through experimentation designed to measure the effectiveness on the usability of security APIs.

The full chart of ancestor–descendant relationships, including empirical validation between papers is shown in (Figure 1). This shows instances of empirical validation and the ancestor–descendant relationships between papers (and the recommendations they provide) across our corpus.

We focus on the 13 usable Security API designer papers and discuss how the majority of the recommendations they provide derive from Saltzer and Schroeder [76], Bloch [10, 11], and Gamma et al. [25]. This shows that these works have become a strong influence on security API design recommendations today. We also find instances where recommendations from the 1975 paper of Saltzer and Schroeder are directly referenced by security API design works of 2020.

4.3.1 Saltzer and Schroeder: Once Upon a Time. In 1975, Saltzer and Schroeder presented eight design principles to help guide the design of protection mechanisms and prevent security flaws [76]. The design principles were *adapted* by a revision of material originally published by Saltzer in 1974 [75].

Saltzer and Schroeder's work has been thoroughly influential through many relationships by works very different from each other, addressing fields from security policies [78] from security for Java applications [28] to cryptographic APIs. Their work has also been *empirically* validated [82]. This level of influence establishes Salzter and Schroeder's work as a strong foundation for security API design recommendations.

Gutmann's release of the Cryptlib cryptographic API in 1995 acted as a gateway between the security engineering recommendations published by Saltzer and Schroeder and 7 of our 13 security API designer papers. Gutmann *adapted* Saltzer and Schroeder's design principles when designing Cryptlib [35].

Gutmann's Cryptlib advertised a *high-level interface* aiming to improve usability while maintaining a strong level of security.

> "Cryptlib provides anyone with the ability to add strong security capabilities to an application in as little as half an hour, without needing to know any of the low-level details that make the encryption or authentication work." [35]

Bernstein et al. built on Cryptlib and presented NaCl, an even more abstracted cryptographic API that *compared* NaCl to Cryptlib in detail [9]. NaCl, itself, has forks such as Libsodium, which also has forks including Monocypher. We can see how strongly Gutmann's *adaptation* of Saltzer and Schroeder's work has influenced the design of cryptographic APIs today.

In 1999, Gutmann carried forward these adaptations when presenting his own set of recommendations to help improve the design of cryptographic security architecture [36]. Gutmann's recommendations were also an *adaptation* of principles used to design NSA's Security Service API. In 2002, Gutmann concludes his trilogy, presenting a set of recommendations in the form of lessons learned from implementing cryptographic software [37]. These recommendations are the oldest of the security API designer papers. Compared to Saltzer and Schroeder, Gutmann's recommendations [37] have not been as widely validated by or related to other works.

The Emergence of Green and Smith's Principles: In 2016, Green and Smith presented 10 recommendations to help developers create more usable and secure cryptographic APIs [31]. The recommendations stemmed from the *distillation* of Gutmann's work and the *adaptation* of a series of API design recommendations defined by Bloch in 2006 [11]. Green and Smith's work is the ancestor, in the ancestor–descendant relationship, to four security API designer papers [1, 56, 64, 87]. Their recommendations are also empirically validated by Patnaik et al. [70], another security API designer paper.

Patnaik et al. [70] offered an *empirical* validation in this chain through evaluating the 10 Green and Smith [31] principles. Through an analysis of over 2,400 Stack Overflow questions and responses from developers facing challenges using seven cryptographic libraries, they found 16 usability issues that were mapped against the 10 principles of Green and Smith [31]. They analyzed the extent to which the 10 principles encompassed the 16 usability issues and also identified additional issues that were not addressed by Green and Smith's principles. Based on this, they derived additional recommendations: *four usability smells*, which are indicators that an interface may be difficult to use for its intended users.

In 2018 Mindermann et al. presented recommendations for designing cryptographic libraries by studying Rust cryptographic APIs [56]. They addressed insecure defaults, authenticated encryption in low-level libraries, lack of warnings about deprecated/broken features, and the scarcity of documentation and example code from low-level libraries. They *compare* their set of recommendations against Green and Smith:

> "Compared to Green and Smith's top ten principles, our recommendations are more specific but do not conflict with their suggestions." [56]

Acar et al. [1] *adapted* Green and Smith to evaluate solutions from 256 Python developers attempting tasks such as symmetric and asymmetric cryptography using one of five different cryptographic APIs.

Oliveira et al. [64] *distill* the work of Green and Smith as well as Acar et al. as part of an empirical study to understand the developer's perspective of API blindspots through a series of code scenarios. Oliveira et al. analyzed the developer's personal traits, such as; perception of correctness, familiarity with code, and level of experience.

Votipka et al. *adapted* Green and Smith to help understand what errors developers tended to make and why. Votipka et al. analyzed 94 submissions of code attempting security problems, resulting in 182 identified unique security vulnerabilities [87].

Votipka et al.'s recommendations from 2020 is the latest in an ancestral chain dating back to Saltzer and Schroeder's design principles of 1975, aimed at protection mechanisms [76]. Authors like Gutmann and Green and Smith played a pivotal role when tailoring Saltzer and Schroeder's design principles toward the security API design recommendations of today [31, 37]. However, it is interesting to identify a direct and contemporary relationship between Votipka et al. and Saltzer and Schroeder. Votipka et al. *borrowed* Saltzer and Schroeder's principles to highlight design violations made by developers introducing too much complexity in their code. This shows two very different forms of evolution; on the one hand, we see security engineering recommendations from 1975 strongly influential and transforming slowly over time to address challenges in niche fields such as the design of cryptographic APIs and security API design recommendations, and on the other hand, Saltzer and Schroeder's principles are still relevant today and flexible enough to address the challenges of designing security APIs directly.

4.3.2 Bloch: Know Your Audience. Learning a new language requires knowing the grammar (how to correctly structure the language), the vocabulary (how to name things you want to talk about), and the common and effective ways in which to say things (usage). These practices are also applicable to programming languages. Many have addressed the first two practices [5, 30]. However, Bloch notes that many Java developers do not have a good understanding of usage. In 2001, Bloch dedicated Effective Java to address the practice of usage. The book offers advice on code structure, and the importance of others' understanding and code readability to improve ease of use when making future modifications [10]. Throughout the book, Bloch evaluates and compares his recommendations to Gamma et al.'s design patterns [25].

> "A key feature of this book is that it contains code examples illustrating many design patterns and idioms. Where appropriate, they are cross-referenced to the standard reference work in this area [Gamma95]." [10]

In 2006, Bloch takes a new direction, *adapting* his recommendations from Effective Java to provide guidance for designing good APIs. Initially Bloch provides this guidance in the form of a presentation at Google Inc. Following this, Bloch condenses the essence of the presentation into 39 recommendations [11]. These recommendations were later adapted in 2016 by Green and Smith to improve the usability of security APIs through design [31].

Bloch's work is also *adapted* by Acar et al., who also *adapts* the works of many, including Green and Smith [31], Henning et al. [40], and Nielsen [60], to compare the usability of Python-based cryptographic APIs [1].

> "We *adapt* guidelines from these various sources to evaluate the APIs we examine." [1]

We thus have an intricate chain of usable Security API designer recommendations—ones that both inform Green and Smith [10, 11], and those that Green and Smith inform [1, 56, 64, 70, 87]. In Bloch's ancestry, we do not find any explicit evidence that Bloch's recommendations have been empirically validated as they moved into Green and Smith, though there is some traceability to Gamma et al.'s design patterns—that are rooted in observations of developers' problem-solving practices. Bloch [10] discusses the many architectural advantages of Gamma et al.'s design patterns, and more specifically Gamma et al.'s factory pattern [25]. This suggests that we need not only further empirical validation of Green and Smith, and the ancestry this work builds on.

4.3.3 Georgiev: The Most Dangerous Code In The World. Georgiev et al.'s work is in itself an empirical study, in that Georgiev et al. provide evidence that the SSL certificate validation is broken in many security-based applications and libraries [26]. Georgiev et al. found that any SSL connection from cloud clients based on the Amazon's E2C Java library are vulnerable to man-in-the-middle attacks and are due to poorly designed APIs of SSL implementations, in turn presenting developers with a confusing set of parameters and settings to decipher. Georgiev et al. conclude their paper by presenting recommendations for both application developers and SSL library developers [26].

Georgiev et al.'s recommendations are *empirically* validated by O'Neill et al. [66], another security API design paper, who also *empirically* validate the works of Brubaker et al. [14] and Fahl et al. [23]. A connection is also seen between Georgiev et al., Brubaker et al. and Fahl et al., as the latter two papers *compare* their work to that of Georgiev et al.

Building on Georgiev et al. work, O'Neill et al. present the **Secure Socket API (SSA)**, a simplified TLS implementation using existing network applications [66]. O'Neill et al. build upon earlier work on TrustBase—an effort to improve security and flexibility available to administrators who select the certificate validation for their applications [65]. SSA presents the administrator with the choice of standard validation or TrustBase. By selecting TrustBase, administrators have finer-grained control over validation. O'Neill et al. analyze the design of OpenSSL, providing recommendations to help improve the design. These recommendations generally apply when designing security APIs.

Meng et al. perform an empirical analysis of StackOverflow posts to understand challenges faced by developers when using secure coding practices in Java [54]. They identify security vulnerabilities in the suggested code of answers provided through StackOverflow. The findings of the study suggests more consideration should be given to secure coding assistance and education, bridging the gap between security theory and coding practices. A comparison is made to Georgiev et al.'s work [26].

> "*Compared* with prior research, our study has two new contributions. First, our scope is broader. We report new challenges on secure coding practices [· · ·]. Second, our

investigation on the online forum provides a new social and community perspective about secure coding. The unique insights cannot be discovered through analyzing code." [54]

Similarly, Meng et al. also *compare* their work to Egele et al., who developed CryptoLint, a static program slicing tool designed to check applications from the Google Play marketplace. Egele et al. find that 88% of these applications that use cryptographic APIs make at least one mistake, such as failing to provide "security against chosen plaintext attacks (IND-CPA) and cracking resistance" [22]. Egele et al. *adapt* the work of Bellare et al. [8] and Desnos' Androguard [21].

"Our tool, called CryptoLint, is based upon the Androguard Android program analysis framework. [· · ·] We adopt the notation used by Bellare and Rogaway." [22]

Not only does the work of Bellare et al. and Desnos provide a foundation for Egele et al.'s analysis, but they directly influence the security criteria used by CryptoLint. Based on the analysis Egele et al. present, a set of countermeasures against the vulnerabilities were found.

Between these four security API designer papers [22, 26, 54, 66], we can see that a good foundation is forming. In particular, Georgiev et al.'s [26] recommendations have been *empirically* validated by O'Neill et al. [66], and *compared* by Meng et al. [54], Brubaker et al. [14], and Fahl et al. [23]. The engagement with Georgiev et al. through many ancestor-descendant relationships and it being an empirical study in itself, cements Georgiev et al.'s work as a strong foundation upon which future security API recommendations can be built by the research community. To ensure this, engagement through further empirical studies and ancestor-descendant relationships should continue.

5 THREATS TO VALIDITY

We identify four main threats to validity.

First, our search terms on Google Scholar, IEEExplore, and ACM DL (see Section 3) may have overlooked some papers relevant to our study. We believe we have mitigated this threat and are confident that no relevant work has been overlooked. Our search was extensive using general search engines and manually checking a subset of key venues. Forward and backward snowballing was used to ensure cited papers at other venues were not missed. Table 3 lists all venues and publishers for our 65 papers providing actionable recommendations.

Second, the categorization was conducted inductively. To ensure the categorization was consistent, we did an independent coding and calculated Cohen's κ [18], demonstrating that our categorization was consistent between coders. However, roughly one-fifth of recommendations have two categories. Cohen's κ is not designed for data with multiple categories, so when calculating interrater reliability, we used only the first categorization. This is unlikely to affect the overall analysis as the κ-value for just the first category is high (0.74)—we would expect a second category to also be consistent.

Third, search neutrality could be a threat as the authors did not search resources in incognito. However, we did cross-check our results from our online search through multiple sources. This included a manual search for relevant papers across key venues. So while there is a small risk that search neutrality would have an impact, we are confident that our search is extensive and unlikely to miss any key works in the field.

Fourth, Our SLR studies 65 papers that present 883 actionable recommendations for improving usability and security. Of these 65 papers, 14 papers have been empirically validated. These 14 papers present a total of 179 recommendations, accounting for 20% of the 883 recommendations.

It is difficult to determine specifically which of the 179 recommendations have been empirically validated, because the papers that perform the empirical studies rarely list out all the recommendations from our 14 papers. There are a few instances where every recommendation is listed out. For example, Patnaik et al. maps each one of the 10 recommendations presented by Green and Smith [31] against 16 usability issues faced by software developers when trying to use seven cryptographic libraries [70].

In 2002, Gutmann studies his own design principles from 1999, on which Cryptlib was designed [36]. As a result of studying his own design principles, Gutmann presented a set of new recommendations in the form of lessons learnt from the process of designing a cryptographic library [37].

However, these are two rare examples where every recommendation presented by an earlier work is listed and empirically studied, one-by-one, by a newer paper. This challenge is also true for the remaining ancestor-descendant relationships. When tracing the ancestry of a set of recommendations presented by a paper, we often found that a paper would say that they "adapted" or "compared" the work of an earlier paper. For example, Mindermann et al. compared their recommendations to the work of Green and Smith [31, 56].

"Compared to Green and Smith's top ten principles, our recommendations are more specific but do not conflict with their recommendations" [56].

Our SLR has made efforts to specifically address and analyse each of the 883 recommendations and introduce some clarity to the subject. By codifying and categorising the 883 recommendations, we present seven categories and 36 sub-categories in Table 4. We also perform an analysis using this data in Table 5. The information shown through Tables 4 and 5 identifies what challenges the 883 recommendations focus on.

6 DISCUSSION

6.1 The Importance of the Longstanding Principles

What makes the works of Saltzer and Schroeder, Bloch, Nielsen, and Gamma et al. referable is that not only are they actionable but also that they are flexible enough to transition into different branches of software engineering and computer security through adaptations [10, 11, 25, 60, 76]. This is no clearer than in Saltzer and Schroeder's case [76], where they define a set of recommendations to address the design challenges of protecting information, stored on computers, from unauthorized access. Gutmann facilitated the transition from security engineering to security API design by using Saltzer and Schroeder's recommendations to design the Cryptlib cryptographic API [35]. Gutmann's work is later related to by Green and Smith, from which many other Security API designer papers grew [31]. This evolution was possible primarily because Saltzer and Schroeder's recommendations were actionable. Saltzer and Schroeder's recommendations are adapted again in 2020 directly by Votipka et al. [87], proving that not only have their recommendations stood the test of time but also that they are still relevant for addressing the challenges faced today by security API designers.

Gamma et al. also play an influential role in the state of today's security API design recommendations. In 2001, Bloch transitions the design pattern's of Gamma et al. to address usability challenges in Java programming [10]. In 2006, Bloch adapts his own work toward designing good APIs [11]. Green and Smith tailor Bloch's API design recommendations for security API designers [31]. The wide-spread influence seen through Gamma et al.'s ancestry explains why it referred to through many ancestor-descendant relationships.

Saltzer and Schroeder permeate several of our categories, but some also come from elsewhere. The *Documentation* category likely has its origins in the work of Nielsen and UI usability [60]. Nielsen's recommendations are adapted by Acar et al. for comparing the usability of Python-based cryptographic APIs, where the importance of good documentation is highlighted. Several other papers have highlighted the importance of usable and high quality documentation [7, 10, 11, 20, 56, 60, 69, 70, 73, 86, 89].

Before going on to advise on how one can ease novice programmers into programming, Pane and Myers [69] quote their guidance:

> "Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large." [60]

Pane and Myers go on to inspire others and bring usability specifically to developers. 20 years later Green and Smith describe their 10 principles for creating usable and secure crypto APIs [31]. For example, "Make APIs easy to use, even without documentation" [31]. Yet again, these earlier usability guidelines have been restated, rediscovered and then returned to.

Eleven recommendations say specifically to use (and occasionally not to use [34]) Gamma et al.'s *Design Patterns* and six reference *Factory Patterns* directly. Four papers related to Gamma et al.'s work, and a further two validated the patterns empirically. Perhaps the easy-to-recall names of many of the patterns have helped cement the work—but whilst we identified recommendations to use design patterns and to document their use, we did not see new versions of the *Factory, Visitor, Observer,* or *Singleton patterns* being restated for Security API designer papers, or other more specialized fields.

This does not mean, however, that the original Gamma et al. *Design Patterns* are not connected to the Security API designer field. The *Design Patterns* have influenced the recommendations from the current literature (e.g., Bloch's), and have become an underlying standard upon which new recommendations are built. The works that presented these longstanding principles can be considered as a set of rules that are widely known to the current software engineering, computer security, and the usable security research communities. Future advances in security API design recommendations can refer to these standards, without hesitation, because these longstanding principles are tried and tested through developing challenges.

6.2 More Validation Please!

How did Saltzer and Schroeder's work from 1975 remain relevant over the past 47 years? Why is a paper from 1994, authored by Nielsen, still influential today? Or why does a series of design patterns written by Gamma et al. in 1993 become part of an SLR written in 2021? The answer to all these questions is seen through empirical validation and our ancestor–descendant relationships. Without the ancestry chain stemming from Saltzer and Schroeder, would Votipka [87] even know their recommendations existed? It is unlikely, which is probably the case for many other design recommendation papers from that time. This is why empirical validation is necessary. The purpose of empirical validation helps set aside poor design recommendations and brings forward recommendations that prove to be effective. Empirical validation provides assurance to designers that the recommendations they are considering do in fact help design better software.

To understand to what extent ineffective recommendations have been propagated over time, one would need to identify the ineffective recommendations, which requires empirical studies to be performed on the set of recommendations presented by the papers discovered through our SLR. We encourage empirical validation to identify the effectiveness of recommendations and prevent the propagation of ineffective recommendations. This is a task for the research community as a whole. We encourage the research community to engage with these works and perform this greater analysis.

Whilst we found that many recommendations have not been validated or related to ($\frac{166}{883}$, 19%), overall the software engineering and security communities seem to be making strides toward it. Yet, this seems to be less so with papers that provide both general and security-focused recommendations for developing APIs, at 39% and 33% of papers, respectively (see Table 7).

Our work shows that 22% of all papers are empirically validated. More should be done to directly engage with the ancestral *chains* deriving from earlier recommendations rather than validating a singular set of recommendations to ensure a depth from earlier works to contemporary papers. When creating new recommendations then, we should be looking at the history of where our knowledge comes from.

Therefore, we argue that studies should not only focus on validating *contemporary* papers but also engage with the earlier and large body of knowledge concerning usability, and its implications for APIs and security. To engage with earlier papers, one could run a standalone empirical validation study on the recommendations presented in these, for example. By applying recommendations in practice through experimentation and providing detailed analyses, one can help build more solid foundations as empirical validation can then be referenced by future studies. Upon this strong foundation, the recommendations can be transformed to create new recommendations specific to fields such as Security API designer guidance.

Many recommendations have arguably changed little from those made 25 or even 50 years ago—yet relatively few of the earlier works are referenced in the ancestral chains we have analyzed.

Modern recommendations are clearly still being inspired by earlier works and to avoid restating ourselves, as a community, we must take earlier, more established guidance and ensure—the foundational principles—are validated fully.

We see evidence of a well-referenced and well-validated paper in the making through the work of Georgiev et al. [26]. A set of recommendations that has been empirically validated and influenced the works of many others, Georgiev et al. has the potential to influence many more in the field of security API design. To ensure this potential, more validation is needed, their recommendations need to be tested against varying conditions and challenges.

6.3 Things to Consider when Designing a Security API

For researchers and engineers who wants to design a new security API, they can start by looking at the recommendations that fall under *Construction* and *Understanding* (Tables 4 and 5). Furthermore, they can trace the ancestry of these recommendations and see the types of ancestor-descendant relationships and empirical validation that have formed these ancestries through Figure 1. If empirical validation has been performed, then they can consider using these recommendations to design their security API, but if not, perhaps they should consider performing their own empirical study to validate these recommendations before using them to design a security API. To further improve the design of their security API, they can look at recommendations from categories that have not had much attention from Security API literature and start by validating them. For example, of the 84 Security API designer recommendations, 14 (17%) fall under *Documentation*. But there are other paper types as well, for example, 40 of the 307 (13%) of Security Engineering paper contribute to *Documentation*. An API designer can look at these recommendations and using Figure 1, look for connections between the literature of these two paper types, or they could see if Security Engineering recommendations are relevant to their Security API design and form a connection through one of the ancestor-descendant relationships or through an empirical study performed by them.

When selecting a set of recommendations to follow, researchers and engineers can benefit from understanding the context under which those recommendations were formed. For example, Bloch presents a series of recommendations specific to Java. Bloch found that the existing literature at

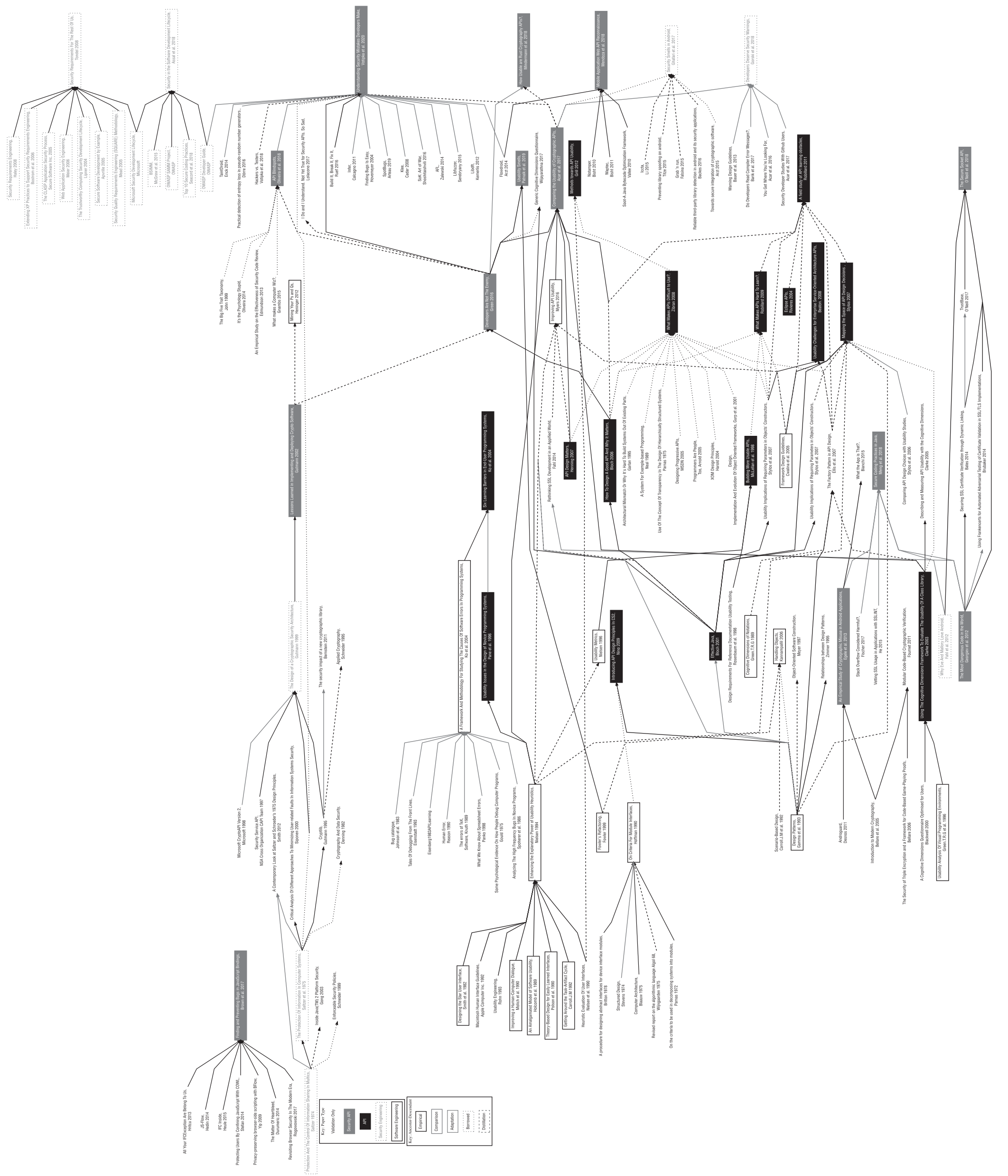

Fig. 1. Graph showing links between papers where one paper has built upon the work of another, and key. These were identified through the paper survey, using Security API designer papers with recommendations to identify and validate how knowledge has been translated.

the time focused on grammar and vocabulary but found there to be a lack of guidance surrounding correct usage when programming in Java [10]. Later, in 2006, Bloch adapts this work to address the challenges of designing a good API [11]. By reading these papers, researchers can not only see the recommendations presented but also the challenges these recommendations address.

Reading the papers can help the researcher determine whether the recommendations are relevant to their specific challenges or not. For researchers who want to evaluate a set of recommendations before using them, they can refer to the paper to help guide their evaluation.

To help guide the design of security APIs, we suggest works that have been empirically validated and their recommendations can be readily used, works that require empirical validation before using their recommendations, and longstanding principles that are still actionable today.

(1) ***"Ready to use" recommendations [31, 36, 37].*** Green and Smith's recommendations, to help improve the usability of security APIs, have been empirically validated by Patnaik et al. [70]. Green and Smith's work has also influenced 5 of the 13 Security API designer papers through the 4 ancestor-descendant relationships [1, 56, 64, 70, 87]. Researchers and engineers can also follow the works of Gutmann, who documented the process of designing the architecture of a novel cryptographic library and empirically validated these design principles resulting in a series of lessons learnt to conclude the trilogy [35–37]. Gutmann's work has also been *adapted* by Green and Smith and introduced to the field of designing security APIs.
(2) ***Security API designer papers in need of validation [1, 13, 53, 54, 64, 66, 70, 87].*** These papers are newer compared to Green and Smith and Gutmann, and so they have not been empirically validated or formed ancestor-descendant relationships with even newer papers. One can argue that because these papers present recommendations that have been influenced by a long ancestry dating back to Saltzer and Schroeder that these recommendations too must be effective. However, we argue that the reason the ancestry became as long as it did is because the works of Saltzer and Schroeder, Gutmann, and Green and Smith were empirically validated, resulting in many works citing and forming ancestor-descendant relationships with these three papers and progressing their recommendations through time [31, 36, 76]. Therefore, we encourage that before using the recommendations of the remaining security APIs designer papers, researchers and engineers should empirically validate these works.
(3) ***Validating corporate literature [15, 55, 67].*** Many of the works categorised as *security engineering papers* are published by corporations such as Microsoft [55]. When tracing the ancestries of our papers, we found that corporate literature presents recommendations not based on earlier works, but instead based on experience and engaging with practical exercises such as training new developers. Most of the recommendations offered by corporate literature fall under the *Organisational Factors* category (Table 4). *Organisational Factors* may be an important aspect of designing a security APIs and it encourages researchers to consider the developers who use security APIs and the training they have and may require. However, this cannot be known unless the effect of *organisational factors* on the design of security APIs is validated.
(4) ***Revisiting the longstanding principles [10, 11, 25, 37, 60, 76].*** Researchers and engineers can benefit from revisiting these works to applying their recommendations to the designing challenges of today. Works such as Saltzer and Schroeder [76] have proved to still be relevant and actionable through Votipka et al. [87].

6.4 Meta-Recommendations

Our categorization of the recommendations are neutral—we do not frame the categories as things one should or should not do—but rather describe what type of advice the recommendations offer. The recommendations discovered through our systematic literature review are relevant to different stakeholders, this includes: the developer, the research community, the company. After analyzing many different recommendations, we offer meta-recommendations based on our extensive analysis of the literature and relevance to the different stakeholders involved.

(1) ***The Importance of Quality Assurance [6, 7, 10, 11, 15, 55, 63, 68, 79]. Relevance: The Research Community.*** Software is not developed in isolation. Have engineers and tools review code to spot rough edges and ensure best practice is followed.
(2) ***Software Engineering Matters [11, 20, 25, 43, 63, 75–77]. Relevance: The Company.*** Performing quality assurance through code reviews is an important aspect of software engineering but there are many other aspects a developer should consider. One should follow best practices for software development and ensure code produced is of a high quality. Give mechanisms for access control and have a plan for how the code will be maintained. Getting good, minimal, well abstracted, well-structured code will pay dividends in the long run.
(3) ***Embed Security at Every Stage [15, 49, 76, 80]. Relevance: The Developers.*** Design security in from the start by compartmentalizing components, and having sensible defaults. Have a plan for dealing with bugs and defects.
(4) ***Show, and Tell [11, 34, 56, 70, 86]. Relevance: The Developers.*** Documentation matters! Document how the APIs work. Document how programmers should use them. Provide exemplars. Standardize as much as possible. Make sure the documentation is easy to find and read.
(5) ***API Developers are not an Island [15, 55, 68, 80]. Relevance: The Company and The Research Community.*** An API might be for programmers to use, but they are often maintained and managed within organizations. Executives need training to make good decisions, and organizations need a plan to develop their security knowledge and practices. API Developers will be influenced by outside forces (be they regulatory, risk-based, or third-party developers).
(6) ***Write a Specification [6, 11, 15, 38, 51, 52, 55, 79, 81]. Relevance: The Developers.*** Break the functionality of a security feature into smaller units. Write a specification to address the first unit. Start by gathering requirements, and update those requirements as new threats are found. Once a unit is implemented move on to the next one.
(7) ***Remember Programmers are Human [7, 17, 31–34, 42, 46, 47, 57, 60, 61, 69, 85, 89]. Relevance: The Developers and The Research Community.*** Improve the readability of code for programmers who have to read it. Draw programmers' attention to the important bits; make it easy to spot mistakes, and to check when they have got it right. Usability is not just for users.

These seven guiding principles summarize our seven categories and bring together much of the advice for developing secure APIs as well as advice for more general software engineering. The synthesis of software guidance shows that security is an engineering challenge as so consequently when analyzing the overall literature, we found that advice for improving the design of security APIs was often a subset of a broader set of recommendations for improving general engineering. An example of this can be see through Gutmann while designing the architecture of Cryptlib [36].

Our meta-recommendations show that when designing a security APIs, the literature sees security as one of many important factors to consider. This is why only one of our seven meta-recommendations addresses security (*Embed Security at Every Stage*), while the others address the design as a whole. For example, *Show and Tell* focuses on improving documentation, which considers the developer's need to make sense of the security API and to clearly understand how to implement its functions. Clarifications through documentation and examples of code implementations can prevent misconfigurations and potential misuse.

Our meta-recommendations are not the sum total of all advice, but they cover what we distilled as a substantial amount with common points that multiple experts and papers have suggested. Many of the papers that have been cited along our meta-recommendations have not yet been empirically validated, but they have influenced many works through our ancestor-descendant relationships. We encourage the research community to engage with these works and perform empirical studies to test the effectiveness of the recommendations presented. Whilst these are abstract and not *actionable* by developers, they should guide broad thinking in both academic and practitioner material. These meta-recommendations are not exhaustive, but provide grounds for future thinking and development of future recommendations to improve the usability of security APIs.

We also note that some papers are referenced by many of the principles: References [11, 25, 60, 76], amongst others. Perhaps then there should be an eighth principle:

(8) ***Build on Longstanding Principles. Relevance: The Developers and the Research Community.*** The recommendations presented by the earlier works to address the challenges of the time are still worth knowing about, because they heavily influence the recommendations for designing the security APIs of today. While more *must* be done to validate these recommendations empirically, the refinement and restatement through the ancestries we have covered, such as Saltzer and Schroeder, Bloch, and Gamma et al., suggest they are still helpful and relevant today.

7 CONCLUSION

Our study is the first to systematically analyze the recommendations that inform Security API designer papers, crossing scientific communities working on security, APIs, and software engineering. Our research questions systematize and learn where recommendations come from, whether they build on validated work, and whether these bring a strong empirical focus to supporting developers with creating usable APIs.

From an analysis of 65 papers guiding developers, including 13 specifically targeted at providing recommendations to developers on how to create usable and secure APIs, and 883 recommendations found within the papers, we identified seven broad categories of recommendations and 36 descriptor sub-categories. These categories and descriptors provide a system for understanding the knowledge we have for guiding developers to produce better code, understand environments, and interface with organizations. The community has made some strides toward validating recommendations, but more must be done within Security API designer literature to improve empirical validation. As we identified, there are different types of ancestry according to their attention to *abstract* and *concrete* recommendations.

Coverage is important alongside validation rates. Through the ancestry analysis, we identified the well established ancestral chains between different areas of literature.

If the new Security API designer recommendations stem from well-validated ancestral chains, then it will be a stronger, more reliable set of Security API designer recommendations as more validation may have been carried out in the chain. This could result in more than one chain originating from historic sets of recommendations.

In addition, further developing work in the area ought to address the earlier literature of the usability field to more appropriately attend to earlier principles and recommendations. This is because, as we identify in our *Meta-Recommendations*, many earlier and well-validated papers address similar contemporary recommendations. Perhaps we do not need to reinvent the wheel so much as assess and renovate the parts to make them roadworthy for usable Security API designer recommendations today.

REFERENCES

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2017. Comparing the usability of cryptographic APIs. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 154–171.

[2] Hikari Ando, Rosanna Cousins, and Carolyn Young. 2014. Achieving saturation in thematic analysis: Development and refinement of a codebook. *Comprehens. Psychol.* 3 (2014), 03–CP.

[3] Amy J. Ko and Brad A. Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Visual Lang. Comput.* 16 (2005), 41–84.

[4] Axelle Apvrille and Makan Pourzandi. 2005. Secure software development by example. *IEEE Secur. Privacy* 3, 4 (2005), 10–17.

[5] Ken Arnold, James Gosling, and David Holmes. 2000. *The Java Programming Language*. Vol. 2. Addison-Wesley Reading.

[6] Hala Assal and Sonia Chiasson. 2018. Security in the software development lifecycle. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS'18)*. USENIX, 281–296.

[7] Jack Beaton, Sae Young Jeong, Yingyu Xie, Jeffrey Stylos, and Brad A. Myers. 2008. Usability challenges for enterprise service-oriented architecture APIs. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 193–196.

[8] Mihir Bellare and Phillip Rogaway. 2005. Introduction to modern cryptography. *UCSD CSE* 207 (2005), 207.

[9] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *Proceedings of the International Conference on Cryptology and Information Security in Latin America*. Springer, Berlin, 159–176.

[10] Joshua Bloch. 2001. *Effective Java*. Pearson Education.

[11] Joshua Bloch. 2006. How to design a good API and why it matters. In *Proceedings of the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. ACM, 506–507.

[12] Gustav Boström, Jaana Wäyrynen, Marine Bodén, Konstantin Beznosov, and Philippe Kruchten. 2006. Extending XP practices to support security requirements engineering. In *Proceedings of the International Workshop on Software Engineering for Secure Systems*. ACM, 11–18.

[13] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and preventing bugs in JavaScript bindings. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 559–578.

[14] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 114–129.

[15] BSIMM. 2009. Retrieved from https://www.bsimm.com.

[16] John M. Carroll and Mary Beth Rosson. 1992. Getting around the task-artifact cycle: How to make claims and design by scenario. *ACM Trans. Info. Syst.* 10, 2 (1992), 181–212.

[17] Steven Clarke and Curtis Becker. 2003. Using the cognitive dimensions framework to evaluate the usability of a class library. In *Proceedings of the 1st Joint Conference of EASE PPIG (PPIG'03)*. EASE, 359–366.

[18] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Edu. Psychol. Measure.* 20, 1 (1960), 37–46.

[19] Krzysztof Cwalina and Brad Abrams. 2008. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Pearson Education.

[20] Jim des Rivières. 2004. Eclipse APIs: Lines in the sand. In *Proceedings of EclipseCon*.

[21] Anthony Desnos et al. 2011. Androguard. https://buildmedia.readthedocs.org/media/pdf/androguard/v3.1.0-rc2/androguard.pdf.

[22] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 73–84.

[23] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 50–61.

[24] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*. Springer, Berlin, 406–431.

[26] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 38–49.

[27] Mohammad Ghafari, Pascal Gadient, and Oscar Nierstrasz. 2017. Security smells in android. In *Proceedings of the IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM'17)*. IEEE, 121–130.

[28] Li Gong and Gary Ellison. 2003. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation* (2nd ed.). Pearson Education.

[29] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. 2018. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic API misuse. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS'18)*. USENIX, 265–281.

[30] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java Language Specification*. Addison-Wesley Professional.

[31] Matthew Green and Matthew Smith. 2016. Developers are not the enemy!: The need for usable security APIs. *IEEE Secur. Privacy* 14, 5 (2016), 40–46.

[32] Thomas R. G. Green. 1989. Cognitive dimensions of notations. *People Comput. V* 5 (1989), 443–460.

[33] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: A "cognitive dimensions" framework. *J. Visual Lang. Comput.* 7, 2 (1996), 131–174.

[34] Thomas Grill, Ondrej Polacek, and Manfred Tscheligi. 2012. Methods toward API usability: A structural analysis of usability problem categories. In *Proceedings of the International Conference on Human-centred Software Engineering*. Springer, 164–180.

[35] Peter Gutmann. 1995. Cryptlib security toolkit: User's guide and manual. https://cryptlib.com/downloads/manual.pdf.

[36] Peter Gutmann. 1999. The design of a cryptographic security architecture. In *Proceedings of the USENIX Security Symposium*. USENIX, 1–16.

[37] Peter Gutmann. 2002. Lessons learned in implementing and deploying crypto software. In *Proceedings of the Usenix Security Symposium*. USENIX, 315–325.

[38] Charles Haley, Robin Laney, Jonathan Moffett, and Bashar Nuseibeh. 2008. Security requirements engineering: A framework for representation and analysis. *IEEE Trans. Softw. Eng.* 34, 1 (2008), 133–153.

[39] 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 *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*. USENIX, 205–220.

[40] Michi Henning. 2007. API design matters. *Queue* 5, 4 (2007), 24–36.

[41] Daniel Hoffman. 1990. On criteria for module interfaces. *IEEE Trans. Softw. Eng.* 16, 5 (1990), 537–542.

[42] Richard Holcomb and Alan L. Tharp. 1989. An amalgamated model of software usability. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*. IEEE, 559–566.

[43] Mathieu Jacques. 2004. API usability: Guidelines to improve your code ease of use. Retrieved from http://www.codeproject.com/KB/usability/APIUsabilityArticle.aspx.

[44] Poul-Henning Kamp. 2014. Please put OpenSSL out of its misery. *ACM Queue* 12, 3 (2014), 20–23.

[45] Thomas George Kannampallil and John M. Daughtry III. 2006. Handling objects: A scenario-based approach. In *Proceedings of the 24th Annual ACM International Conference on Design of Communication*. ACM, 92–98.

[46] Andrew J. Ko and Brad A. Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *J. Visual Lang. Comput.* 16, 1–2 (2005), 41–84.

[47] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Proceedings of the IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[48] J. Richard Landis and Gary G. Koch. 1977. An application of hierarchical Kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics* 33, 2 (1977), 363–374. Retrieved from http://www.jstor.org/stable/2529786.

[49] Steve Lipner. 2004. The trustworthy computing security development lifecycle. In *Proceedings of the 20th Annual Computer Security Applications Conference*. IEEE, 2–13.

[50] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. 1998. Building more usable APIs. *IEEE Softw.* 15, 3 (1998), 78–86.

[51] Nancy R. Mead and Ted Stehney. 2005. *Security Quality Requirements Engineering (SQUARE) Methodology*. Vol. 30. ACM, New York, NY.

[52] J. D. Meier. 2006. Web application security engineering. *IEEE Secur. Privacy* 4, 4 (2006), 16–24.

[53] Abner Mendoza and Guofei Gu. 2018. Mobile application web API reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'18)*. IEEE, 756–769.

[54] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure coding practices in Java: Challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 372–383.

[55] Microsoft. 2002. Security Development Lifecycle. Retrieved from https://www.microsoft.com/en-us/sdl.

[56] Kai Mindermann, Philipp Keck, and Stefan Wagner. 2018. How usable are rust cryptography APIs? In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'18)*. IEEE, 143–154. https://doi.org/10.1109/QRS.2018.00028

[57] Rolf Molich and Jakob Nielsen. 1990. Improving a human-computer dialogue. *Commun. ACM* 33, 3 (1990), 338–348.

[58] Brad A. Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.

[59] 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*. ACM, 935–946.

[60] Jakob Nielsen. 1994. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 152–158.

[61] Jakob Nielsen. 1996. Usability metrics: Tracking interface improvements. *IEEE Softw.* 13, 6 (1996), 1–2.

[62] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 249–256.

[63] Jaime Niño. 2009. Introducing API design principles in CS2. *J. Comput. Small Coll* 24, 4 (2009), 109–116.

[64] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois A. DeLong, Justin Cappos, and Yuriy Brun. 2018. API Blindspots: Why experienced developers write vulnerable code. In *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS'18)*. USENIX, 315–328.

[65] Mark O'Neill, Scott Heidbrink, Scott Ruoti, Jordan Whitehead, Dan Bunker, Luke Dickinson, Travis Hendershot, Joshua Reynolds, Kent Seamons, and Daniel Zappala. 2017. Trustbase: An architecture to repair and strengthen certificate-based authentication. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. USENIX, 609–624.

[66] Mark O'Neill, Scott Heidbrink, Jordan Whitehead, Tanner Perdue, Luke Dickinson, Torstein Collett, Nick Bonner, Kent Seamons, and Daniel Zappala. 2018. The secure socket API: TLS as an operating system service. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX, 799–816.

[67] OWASP. 2012. OWASP Developer Guide. Retrieved from https://www.owasp.org/index.php/Category:OWASP_Guide_Project.

[68] OWASP. 2012. OWASP SAMM Project. Retrieved from www.owasp.org/index.php/OWASP_SAMM_Project.

[69] John F. Pane and Brad A. Myers. 1996. *Usability Issues in the Design of Novice Programming Systems*. Technical Report. Carnegie-Mellon University.

[70] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. 2019. Usability smells: An analysis of developers' struggle with crypto libraries. In *Proceedings of the 15th Symposium on Usable Privacy and Security (SOUPS'19)*. USENIX, 245–257.

[71] Peter G. Polson and Clayton H. Lewis. 1990. Theory-based design for easily learned interfaces. *Hum.-Comput. Interact.* 5, 2 (1990), 191–220.

[72] Carol Rivas. 2012. Coding and analysing qualitative data. *Research. Soc. Cult.* 3 (2012), 367–392.

[73] Martin P. Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE Softw.* 26, 6 (2009), 27–34.

[74] Martin P. Robillard and Robert Deline. 2011. A field study of API learning obstacles. *Empir. Softw. Eng.* 16, 6 (2011), 703–732.

[75] Jerome H. Saltzer. 1974. Protection and the control of information sharing in Multics. *Commun. ACM* 17, 7 (1974), 388–402.

[76] Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[77] Santonu Sarkar, Girish Maskeri Rama, and Avinash C. Kak. 2006. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Trans. Softw. Eng.* 33, 1 (2006), 14–32.

[78] Fred B. Schneider. 1999. *Enforceable Security Policies*. Technical Report. Cornell University.

[79] R. Seacord. 2018. Top 10 secure coding practices. Retrieved from https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices.

[80] Secure Software, Inc 2005. *The CLASP Application Security Process*. Secure Software, Inc.

[81] Guttorm Sindre and Andreas L. Opdahl. 2005. Eliciting security requirements with misuse cases. *Require. Eng.* 10, 1 (2005), 34–44.

[82] Mikko T. Siponen. 2000. Critical analysis of different approaches to minimizing user-related faults in information systems security: Implications for research and practice. *Info. Manage. Comput. Secur.* 8, 5 (2000), 197–209.

[83] David C. Smith, Charles Irby, Ralph Kimball, and Bill Verplank. 1982. Designing the star user interface. *BYTE* 7, 11 (April 1982), 242–282.

[84] Richard E. Smith. 2012. A contemporary look at Saltzer and Schroeder's 1975 design principles. *IEEE Secur. Privacy* 10, 6 (2012), 20–25.

[85] Jeffrey Stylos and Brad Myers. 2007. Mapping the space of API design decisions. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'07)*. IEEE, 50–60.

[86] Inger Anne Tondel, Martin Gilje Jaatun, and Per Hakon Meland. 2008. Security requirements for the rest of us: A survey. *IEEE Softw.* 25, 1 (2008), 20–27.

[87] 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 *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, USENIX, 1–18.

[88] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14).* ACM, New York, NY, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[89] Minhaz Zibran. 2008. What makes APIs difficult to use. *Int. J. Comput. Sci. Netw. Secur.* 8, 4 (2008), 255–261.

Received 8 December 2021; revised 10 June 2022; accepted 15 July 2022