This electronic thesis or dissertation has been downloaded from Explore Bristol Research, http://research-information.bristol.ac.uk

Author:
Patnaik, Nikhil

Title:
Usable Abstractions for Secure Programming

A Mental Model Approach

General rights
Access to the thesis is subject to the Creative Commons Attribution - NonCommercial-No Derivatives 4.0 International Public License. A copy of this may be found at https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode This license sets out your rights and the restrictions that apply to your access to the thesis so it is important you read this before proceeding.

Take down policy
Some pages of this thesis may have been removed for copyright restrictions prior to having it been deposited in Explore Bristol Research. However, if you have discovered material within the thesis that you consider to be unlawful e.g. breaches of copyright (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please contact collections-metadata@bristol.ac.uk and include the following information in your message:

•Your contact details
•Bibliographic details for the item, including a URL
•An outline nature of the complaint

Your claim will be investigated and, where appropriate, the item in question will be removed from public view as soon as possible.

Usable Abstractions for Secure Programming

A Mental Model Approach

By

NIKHIL PATNAIK

Faculty of Engineering
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Engineering.

MARCH 2023

Word count: sixty thousand, nine hundred and eighty seven

Abstract

People share impressive amounts of sensitive data through the internet. They purchase things online, communicate via mobile phones, store, share and utilise data kept on remote servers. Technological advances, such as the proliferation of mobile devices, and the drive for increased functionality, the advent of the Internet of Things for example, will further strengthen our dependency on the internet. By the end of 2023, it is projected that the number of connected devices world-wide will reach just over 15 billion and it is estimated to increase to almost 30 billion by 2030 [153].

Consider, for example, the proliferation of mobile phone applications. By the end of 2022 there were around 2.5 million application in Google Play app store and about 2.2 million applications in Apple store [117, 152]. Some of these mobile phones incorporate components that deal with sensitive user data, e.g., credit card numbers. Given the rate at which these applications are being developed and the opportunity for catastrophic failures that a networked world entails [159], we need to hold applications against high security standards. A recent study has found that 88% of some 11,000 Android applications show signs of cryptographic misuse through violations of fundamental best practices and standard recommendations.

Many works, within the field of usable security research, have concluded that the reason for the cryptographic misuse seen in these applications is that developers find cryptographic APIs (interfaces through which developers write secure code) difficult to use. The thesis presents a thematic analysis of over 2,400 posts from StackOverflow of questions and responses raised by developers as they try to use 7 cryptographic APIs. The analysis results in the identification of 16 usability issues, categorised into 7 themes. We map the 16 usability issues against a well-known set of principles, defined by Green & Smith. Green & Smith proposed these 10 principles as guidance for improving the usability of security APIs. Through the mapping, we see to what extent the proposed principles are effective and address the 16 identified usability issues.

When it comes to guidance for the improving the usability of security APIs, Green & Smith, is a integral piece of work, but not the only one. We perform a systematic literature review of 65 papers that offer 883 recommendations, spanning over 47 years. Through this literature review, we find out what current recommendations focus, how they came to be, and to what extent have they been validated. We also present a set of meta-recommendations based on our insights from the literature review.

Later in the thesis, we task a study group of 20 participants to design a system through which two parties can communicate securely while achieving the conditions for which the concept of Public Key Cryptography was originally developed. Through our analysis, we elicit the developers' mental models surrounding Public Key Cryptography and identify misalignments between their mental models and the way in which Public Key Cryptography actually works. The misalignments found through the elicited mental models give us a clear understanding as to why developers find it challenging to use cryptographic APIs and also offers insights into how they can be helped in writing secure code.

To conclude, we present a shift-left initiative in CryptoBridge, that introduces the need for cryptographic APIs earlier during the design stage of the Secure Software Development Lifecycle.

Dedication and acknowledgments

As I reflect on this thesis and the years I have dedicated to it, I can now fully appreciate the opportunity that I was given all those years ago by Professor Awais Rashid and Professor Bogdan Warischi. At the time, although I thought I knew what a PhD would entail, in a much more real sense, I didn't and so if it wasn't for the guidance and support provided by Professor Awais Rashid and Professor Bogdan Warischi, this thesis would not be.

Every one of my works that has either become published or a chapter in this thesis is the result of several drafts refined over time due to the invaluable guidance provided by Professor Awais Rashid. Of all the research skills that I have learned under the tutelage of Professor Awais Rashid, the skill that I value the most is the ability to be articulate. It is a very difficult skill to master and I am far from it but during the course of my thesis, I have been able to carve out my own style of writing, for which I owe Professor Awais Rashid a great debt of gratitude.

During my first year, Professor Bogdan Warischi invited me to attend the MSc Cryptography course. The course equipped me with the knowledge to confidently research cryptographic libraries and ask more impactful research questions, ultimately strengthening the quality of all my studies since then. I would like to take this opportunity to thank Professor Bodgan Warinschi for not only inviting me to attend the course and taking the time to clear any doubt I had about the lectures as it greatly improved the quality of this thesis.

I would like to thank Dr.Joseph Hallett and Dr.Andrew C. Dwyer who worked with me on the mountain of a study in the systematic literature review [125]. Dr. Andrew C. Dwyer worked with me for several weeks of this project while we refined a series of categories to accurately capture the commonalities amongst open codes, which eventually became Table 2.4. Dr. Joseph Hallett has overseen and supported my work from the beginning. During this time, we have worked on five papers that have been published in a number of venues [29, 76, 116, 125, 126].

The Bristol Cyber Security Group, as a whole, has been very supportive during these years. I have had the opportunity to meet many bright minded researchers within the group who remind me how vast the subject of cybersecurity is through their research spanning from topic such as privacy to cybercrime to cyberphysical systems and many more. I feel privileged to be a part of this group and to work with the people in it.

I'd like to thank my family, not just for the PhD years but for all of it. You have been there to go for walks during writer's blocks, for coffees during all-nighters, you've let me sleep through my alarm almost everyday (I'll keep working on that) but there have been good days too. This is your thesis as much as it is mine. With all my love - Nikhil

P.s. A special thanks to my little sister just for being you!

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: NIKHIL PATNAIK, BSC (HONS)
DATE: MARCH 27TH, 2023

TABLE OF CONTENTS

LIST OF TABLES

List of Figures

Chapter 1

Introduction

Society, today, strongly relies on a digital economy in the form of a global network infrastructure of digital technologies that supports economic activities, financial transactions, and instant communication. Our trust in these technologies is based on the assurances provided about the effective and secure management of our data. By trusting these technologies, we in turn, place our trust in the secure application of cryptography within.

Yahoo is a web service provider similar to Google in the sense that its users are given email accounts that are used for electronic communication. Email is vital for for communication in the 21st century and a multitude of other online accounts are connected to emails like Facebook, Instagram, and other social media platforms. So compromising our emails in a sense compromises all these other accounts. Emails often hold very confidential and important information and so potentially compromising these emails will cause irreversible harm to the victims. In 2014, Yahoo noticed a potential data breach with attackers gaining access to the traffic that flowed in and out of their servers. However, it wasn't until August, 2016, when the full scale of the attack became apparent. In August 2016, it was reported that a hacker known as 'Peace' attempted to sell information from 200 million Yahoo accounts breached from the attack in 2014. After further investigation, Yahoo revealed that a total of 500 million accounts had bee breached becoming the biggest public breach known worldwide. The data consisted of email addresses, names, telephone numbers, dates of birth, and passwords. Although these passwords were reportedly secured, the hashing algorithm used to secure these passwords was the MD5 algorithm, a weak algorithm that can be easily broken [110]. This security breach had a huge impact on Yahoo both financially and through a tarnished reputation and loss of trust with the public.

This should not be surprising however. Developers are rarely both skilled software engineers and knowledgeable cryptographic experts, even more so with the rise of hobbyist developers developing and deploying mobile and web applications to potentially millions of users around the world. Yet, they have to make decisions that require expertise at each level of the cryptographic architecture that makes the application's security. They need to select which cryptographic primitives to use, how to combine several cryptographic building blocks to obtain more advanced functionality and stronger guarantees, and ultimately have to integrate protocols themselves to fit together within the higher-level applications that employ them. In 2013, a study found that 88% of some 11,000 Android applications consisted of at least one common cryptograhic mistake in its implementation when measured against standard cryptographic practices [46]. In 2012, another study demonstrated how SSL certificate validation was broken in many data-critical applications such as Amazon's and PayPal's merchant SDKs accountable for transmitting financial data from e-commerce sites to payment gateways [62]. The study revealed that 1,074 (8%) of the applications contained SSL/TLS code vulnerable to MITM attacks [62]. Furthermore, a survey was conducted to evaluate the developer's perception of certificate warnings, showing that 50% of the 754 participants could not correctly judge whether the application was protected by SSL/TLS or not [51]. As of 2020, Android users could choose from 3.14 million applications from Google Play [151].

Consider the implementation of a simple client-server application, for example, where the client and server exchange information and where it is desired that the communication stay secret and not known to third parties. A hobbyist developer, inexperienced in cryptography, may be under the impression that a secure communication is confirmed after seeing 'https://' as a prefix or a padlock sign in the address bar. This developer will aim to achieve this confirmation through their implementation. A knowledgeable developer, on the other hand, knows that to establish a secure connection, one needs to build a cipher suite. A cipher suite consists of a number of cryptographic primitives, each chosen to address a specific aspect of the secure connection. Both the cipher suites below achieve the 'https://' indication, and for the inexperienced developer either one would suffice as confirmation that the connection is secure.

1. *A Weak Cipher Suite*: SSL_RSA_WITH_RC4_128_MD5
2. *A Strong Cipher Suite*: TLS_ECDHE_ECDSA_WITH-_AES_128_GCM_SHA256

However, the experienced developer knows that one cipher suite is stronger than the other. For example, it is bad practice to use the MD5 algorithm to hash passwords, simply because the algorithm is too fast [158]. The experienced developer will choose the SHA256 algorithm instead as it is slower, which in turn means an attacker will be slowed down, giving more time to change passwords and patch the vulnerability. A recent study observed 256 Python developers while they were tasked with implementing common cryptographic processes using five cryptographic libraries. The five cryptographic libraries included *cryptography.io*, *Keyczar*, *PyNaCl*, *M2Crypto*, and *PyCrypto* and these libraries were selected based on their popularity and usability. 20% of the implementations that appeared to be functionally correct were deemed secure by the developers, when in fact they were not [1]. Developers find it challenging to define the cryptographic architecture of their application. But what is more dangerous is the developer who believes they have correctly secured their application, when in reality they have not [1, 76]. A developer's competence, when it comes to securing applications, is considerably dependent on their perception of cryptography and how it is presented through code.

A cryptographic library is the medium through which theoretical concepts of cryptography become applicable in software development. A cryptographic library like OpenSSL, for example, offers implementations for a wide range of cryptographic algorithms. A developer accesses the cryptographic library through a cryptographic API. A cryptographic API (Application Programming Interface) is an interface designed by an API designer with a high-degree of cryptographic expertise to help developers implement the cryptographic framework needed to secure their application. When designing a cryptographic API, the API designer assumes the developer's competence. OpenSSL is a low-level cryptographic API, exposing the developer directly to the wide selection of cryptographic algorithms and functions of the library. Unless the developer is also a cryptographic expert like the API designer, they will not know what cryptographic functions they need to implement to secure their application, they will not be able to tell cryptographic algorithms apart based on strength. Ultimately, unless the developer also has cryptographic expertise, they will find the low-level cryptographic API challenging to use, maybe even unusable [85]. Furthermore, the combination of a low-level cryptographic API and an inexperienced developer can lead to cases of cryptographic misuse.

This challenge gave rise to the advent of usable cryptographic APIs. Bernstein designed NaCl, a library with a high-level cryptographic API, as a response to the usability issues reported with the OpenSSL cryptographic API [15]. NaCl presents the developer with a more abstracted representation of cryptographic functions. For example, a process like Public Key Authenticated Encryption requires several steps to be accomplished. An experienced developer may be comfortable using a low-level API, like OpenSSL, to implement these steps and directly engage with the cryptographic primitives, whereas an inexperienced developer may not. The crypto_box API, provided by NaCl, encapsulates these steps and presents them as one, unburdening the developer. There are now several cryptographic APIs that are built on the principle of abstraction, with the aim of being usable.

But usable for whom? LoIacono et al. proposed a classification of security APIs based on their level of abstraction. The classification split security APIs into 2 main categories. APIs with a low-level abstraction, denoted as *security primitives* and APIs with high-level abstraction, denoted as *security controls*. Security APIs of the security primitive class are mostly comprised of cryptographic APIs. LoIacono et al.'s description for security primitives is stated below:

> "This end of the spectrum is made up by foundational means that can be used to realise basic security services such as confidentiality, integrity, authenticity and non-repudiation. These APIs provide a high flexibility since they allow selecting, initialise and combine the associated primitives to specific security controls as needed. This flexibility, however, demands a high degree of knowledge and expertise from the developers. Otherwise they will fail developing robust and effective security protection means." [82]

Security APIs that provided means for secure communication or storage fell under the security controls category.

> “The APIs containing to this category are less flexible than the ones belonging to the security primitives. On the other hand, they are more goal-oriented and ready to use. When implemented correctly, they encapsulate a lot of security expertise and know-how to lift this burden from the shoulders of the developers using a security controls API.” [82]

LoIacono et al.’s definitions can be further extended to cryptographic APIs as well. OpenSSL offers developers with a wide selection of cryptographic algorithms and functions. Due to the low-level design of the API, the developer needs to have a high degree of cryptographic expertise to know what cryptographic primitives to use for a particular task, the difference in strength when comparing different cryptographic algorithms, the parameters to use when calling a function. On the other hand, developers who use a high-level cryptographic API, like NaCl, are relieved of the responsibility of making these decisions as NaCl’s crypto_box API is comprised of pre-selected cryptographic primitives that the API designer deemed appropriate for securing applications. Implementing the concept of abstraction into cryptographic API design introduces a trade-off between flexibility and usability. A low-level cryptographic API allows more flexibility by allowing the developer to directly engage with the cryptographic primitives. This may appeal to experienced developers, but may prove to be too challenging to use for inexperienced developers. A high-level cryptographic API aims to improve usability. This may appeal to inexperienced developers, but it may not be as inviting to an experienced developer due to its lack of flexibility. The experienced developer cannot directly engage with the cryptographic primitives through an abstracted API design, and so a sense of flexibility is lost [82].

So what do we have? We have teams of API designers who offer a number of cryptographic primitives and functions they believe developers find useful through APIs abstracted to a level they believe achieves usability for most, if not all developers. The API designer makes these decisions based on a combination of 2 factors: the first being the API designer’s mental model of cryptographic processes; the second being the API designer’s assumption of the developer’s perception and competence to understand and execute cryptographic processes. The developer has their own mental model of cryptographic processes. The term *mental model* was first used by Craik in 1952 in his book titled ‘The nature of explanation’.

> “A mental model is a physical working model which works in the same way as the process it parallels.” [33]

Here is where the root of the problem lies. The API designer and the software developer have different mental models from each other, when it comes to cryptography. An API designer may have a very low-level understanding of cryptographic processes and can give a very detailed explanation of the underlying working. On the other hand, a cryptographically inexperienced developer will have a much higher-level of understanding, if any, of how the cryptographic processes work. This is a challenging task for the API designer, who adopts abstraction to find a middle-ground between their API and the software developer’s mental models.

Existing research has shown that vulnerabilities arise in software due to misunderstandings of how an API functions [109] or unintentional misconfigurations of security parameters [50, 51]. However, little is understood about the software developer's mental models that lead to such issues and the misalignment between these mental models and the actual guarantees offered by the security API and the underlying assumptions, shown through the API design, about how it should be used by software developers. The problem is that the mental models used by the API designer, seen through their cryptographic API, may differ from the software developer's mental models. The impact of this misalignment between the mental models of the API designer and the developer is seen through the instances where the developer struggles to use the cryptographic API, is confused by the supporting documentation, and in turn misuses the cryptographic functions, leading to potential vulnerabilities being introduced into their software applications.

1.1 Limitations of the State of the Art

The aim of this thesis is to develop usable abstractions for secure programming using a mental model approach. However, in order to suggest a solution, one needs to understand the extent of the problem and to which degree advancements have already been made. The landscape is split into 2 fields of research. Firstly, we look at the research surrounding usability issues associated with security APIs. Secondly, we introduce mental models and then look at the advances that have been made in security research with the use of mental models.

1.1.1 A Mental Model Approach

1.1.1.1 An Introduction to Mental Models

The purpose of mental models is to gain an understanding of an individual's thought process while they engage in dynamic decision making. Mental models comprise of, at least, 3 distinguishable sub-models: an ends model (goals), a means model (strategies, tactics, policy levers), and a means/ends models (connections between ends and means) [4].

The Ends model is defined by the perceptions about what one is trying to achieve through a decision or a stream of decisions over time. The individual can set out to accomplish a series of proximal goals or one ultimate goal.

> "An individual's end model in a dynamic decision making task comprises of this rich set of local-to-global goals." [132]

The Means model encapsulates the strategies, the tactics the individual develops. The individual reflects upon a plan of action, or several plans of action by intuition or an analytical approach, finally selecting one they believe advances towards the perceived goal [103].

The Means/Ends model represents the feedback structure of a dynamic system. The ever changing relationship between the goals and the plan of actions to achieve these goals is conceptualised within the Means/Ends model. If one looks deeper into the literature of mental models, one sees that

Means/Ends models lay on a continuum ranging from ‘operator logic’ to ‘design logic’, concepts defined by Forrester et al., building on the work of Montmollin et al. in the field of human/computer interaction [38, 54].

Are goals perceived? Are they clear? Do they match the observers’ vision of the end or goal to strive for? Do they conflict? [4, 60, 136]. What kind of actions are possible? How does the research limit or expand the individual’s understanding of the actions at their disposal? [132]. These are the types of questions we discuss in more detail, along with the feedback theory, in Chapter 4: *Developer Mental Models of Public Key Cryptography*.

The feedback theory is regarded as a classic view on the concept of perception. The theory proposes that human decision making is a cybernetic loop involving the state of the system, the perceived state of the human, goals, planned action, and action. Of these concepts, 3 are mental constructs: the perceived state, goals, and the planned action. The fourth mental construct is the implicit ‘cognitive model’ of the way the system is structured, resulting in the plan of actions selected [132].

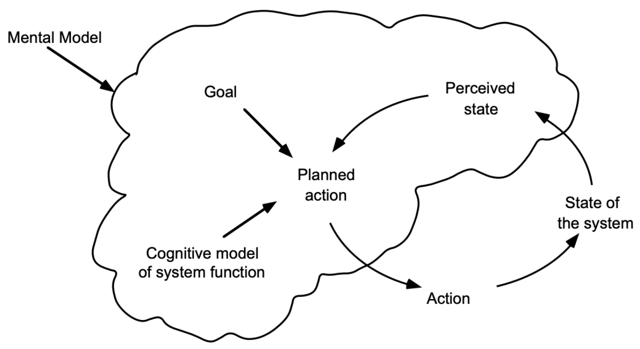


Figure 1.1: This figure shows how the different constructs of the cybernetic loop relate to one another. The loop is comprised of the system state, the perceived state, planned action, and action [132].

System dynamics literature on mental models tend to gravitate towards researching the perceptions of a system (the means/ends model). The grounding hypothesis is:

> “Truer perceptions of system structure lead in the direction of more effective policy decisions and system management.” [132]

Richardson et al. notes that the other constructs of the mental model also influence the result of research associated with mental models. System modelers further endeavored to improve their understanding of mental models, adopting learning’s from subjects like psychology. The feedback theory is strengthened by integrating it with the Brunswikean lens model, a theory from psychological literature to help understand the individual’s ‘perceived state’ [24]. This is the extent to which we delve into the origins of mental models. Now we see how they have been applied in cybersecurity.

1.1.1.2 Existing Research in Security Through Mental Models

Here, we introduce works that elicit and gather insights from the mental models of end users. While these studies focus on the mental models of end users, such as the home computer user or Android and iOS phone users [14, 165], we apply a mental model approach to software development. We aim to understand the perceptions developers hold about cryptographic concepts and the cryptographic tasks they perform. We also highlight how these perceptions differ from that of the API designer, who represents the cryptographic concepts through their own mental models seen through their cryptographic API design. To help structure these studies, we begin asking *open research questions* tailored towards eliciting the developers' mental models and the insights we can gain. The placement of these open research questions indicates the works that have influenced our studies and the types of research questions they are designed to answer.

With the aim of exploring user perceptions of what encryption is and how it is used in everyday life, Wu et al. ran 19 semi-structured phone interviews using a combination of standard interview techniques and diagramming with participants where they described their perception of the encryption process. They identified 4 mental models of encryption that, although varying in the level of complexity, ultimately fall under models of limited access and resemble the concept of symmetric encryption. The study also revealed that the participants believed tasks like encryption are primarily performed by developers and service providers, and that the personal use of encryption is considered somewhat illicit or immoral, or even an indication of a paranoid individual. To encourage the participants, they were presented with a number of use cases, for example, the encryption of smartphones, or the sight of 'https://' in the address bar [168]. This approach ties closely to the concepts presented by Richardson et al. [132]. The Ends model is represented by the task to diagram and discuss the encryption of a smartphone; this is the goal. The participants are familiar with smartphones and so they know the plans of action they can take to achieve the goal, like setting a pin-code for example; this is the Means model. Their perception, shown through the diagram exercise and interview, defines the connections they make between the goal and the selected plan of action. Wu et al. noted that, because the mental models closely associate symmetric encryption, asymmetric encryption is non-intuitive.

Open Research Questions:

- What are the developers' mental models surrounding integral cryptographic concepts like Public Key Cryptography?
- What are the misconceptions seen in the developers' mental models and how do they deviate from the cryptographic concept?

Through a literature review of the mental models used in security and privacy, Camp identified 5 mental models that are used as a means to communicate security and privacy risks [26].

- *Physical Security Model*: E.g. models based on the lock metaphors.
- *Medical Models*: E.g. models based on the phrase 'infected by a virus'.

- *Criminal Models*: E.g. models based on the idea of being arrested if you hack into a system.
- *Warfare Models*: E.g. models based on intrusion detection and firewall tools.
- *Market Models*: E.g. models based on people losing money.

Camp makes an argument for the benefits of using mental models along with the flaws that the use of mental models bring. The use of mental models can reduce miscommunication between people an expert and an everyday person, e.g. a doctor explaining a health risk to a patient [26]. The issue with using mental models, however, can be seen in the scenario of when an expert presents a mental model of a concept through a metaphor based on their own understanding. Counterproductive implications could be drawn from this metaphor. Camp points out that for all its flaws, mental models is still a powerful tool which has yet to be embraced by computer security experts [26].

Open Research Questions:

- Once the developers' mental models have been elicited, how do they compare to the abstractions seen through the design of existing cryptographic APIs?
- Do the developers' mental models exhibit any counterproductive implications?

Wash performed an interview-based study [165] through which he identified 8 folk models of security threats. These security threats influence the decisions of the home computer user when it comes to choosing security software and following security advice. These folk models showed how the home computer users justified their decisions to take backups or not, or to install anti-virus software, or even take on board security advice. These folk models represented false beliefs held by computer users that could leave them vulnerable to the reality of attacker motivations. For example, some of these users believed that viruses could only be caught by visiting bad parts of the Internet. Some users also believed that hackers would only go after data stored in large databases that were associated with large companies, and that individuals like themselves were too insignificant and not worth the trouble. Wash concluded that informing home computer users on how virus-protection software works could encourage them to use it. Wash identified the following folk models [165]:

- *Virus generally bad* but only high level understanding.
- *Virus causes mischief* i.e. annoying problems with computer/data.
- *Virus intentionally downloaded in buggy software* computer will misbehave at some point: e.g. crash or does not boot anymore.
- *Virus supports crime* e.g. by stealing personal/financial information.
- *Hackers perceived as geeks* who want to impress friends.
- *Hackers are criminals* who target big fish (rich and important people).
- *Hackers support crime*, looking for large databases of information.
- *Hackers are burglars* who steal personal/financial information.

Open Research Questions:

- Does the developer show signs of an 'attacker mindset' when designing security protocols?
- If so, what are the threats raised by these developers and how do they mitigate them?

Kauer et al. found 3 more folk models to add to the 8 already identified by Wash [87, 165]. These additional folk models included; *Viruses are Governmental Software*, *Hackers are Governmental Officials*, and *Hackers are Stakeholders with individual and opportunistic purposes* such as Anonymous [87]. Ideas of the government being involved with security threats is also a common conception found by Furman et al. [58].

Dourish et al. view security as a practical problem rather than a technical problem. For security to be effective, in practice, Dourish et al. explained that it was up to the the people, the end-users, to determine whether the configuration of technologies available to them offered enough security to achieve the task at hand [45]. Through a set of interviews, they found 4 broad classes of threat that people often brought up in discussion [45].

- *Hackers* people out to cause mischief and harm. Generally, skilled and motivated by the need to vandalise.
- *Stalkers* might use information available online to pursue an offline threat.
- *Spammers* are people that advertise through unsolicited messaging, wasting people's time and using up organisational resources through an implicit denial of service.
- *Marketers* people who invade privacy by collecting information about activities, purchasing patterns, and so forth.

Weirich & Sasse made the argument that password mechanisms and users formed a socio-technical system, which heavily depended on the users' willingness to take caution and make the effort that security-conscious behavior requires [166]. To encourage security-conscious behavior, users need to be persuaded through policies, tutorials, and training. Weirich & Sasse ran interviews to understand the beliefs held by these users about who was trying to get into their accounts and what are the motivations for doing so [166]?

- *Kids* curious kids who want to see if they can break into a system with the motivation of proving that they can do it. They do not cause serious harm.
- *Vandals* described as 'just plain mad' who can cause serious damage in systems they break into.
- *Criminals* are people who carry out activities related to online banking.
- *Vengeful People* people who are vengeful against a specific individual or want revenge against an organisation.
- *Others* describes industrial spies, terrorists, and jokers.

Open Research Questions:

- What is the developers' thought process when asked to design a security protocol?
- How does the developers' thought process impact their mental models?

Friedmann et al. ran 72 interviews, including a drawing exercise, where participants were asked how they determine whether or not a connection is secure. Participants were also asked what the term 'secure connection' meant to them. Based on this study, Friedmann et al. identified 3 mental models. Participants said that they would look for certain identifiers, to determine a secure connection, such as: a 'https://' prefix in the address bar, a lock/key icon, the stage of the transaction (main page may not be secure whereas payment page should be), the type of information requested, and the type of webpage. Based on this, Friedmann et al. elicited these mental models from the participants explanations.

- *Transit* protecting the confidentiality of information while it moves between nodes.
- *Encryption* specific mechanism for encoding/decoding messages.
- *Remote Side* protecting data once it arrives to the recipient.

Benenson et al. elicited the mental models of smartphone users when interviewed about privacy concerns and awareness [14]. Through the study, Benenson et al. identified 2 mental models: the *Android* and the *iPhone* mental model. The study showed that Android users were more privacy concerned and aware, as data privacy played an important factor when selecting a smartphone. iOS smartphone users were less aware about how their data was being used by applications and were under the impression that even if applications needed access to data, the applications would not necessarily ask for permission from them. Benenson et al. concluded that users who own an Android smartphone would be more likely to care about technical features, whereas users more interested in technology were more likely to own an iPhone [14].

Volkamer & Renaud presented a literature review of the work that has been done in cybersecurity, where the authors of these works adopted a mental model approach [163]. They introduced mechanisms by which mental models can be modified, as they noted that mental models are of a dynamic nature not static. A number of researchers used interviews as a means to encourage the participants of their study to verbalise their mental models [19, 127]. Volkamer & Renaud note the occurrences and categorise them as think-aloud experiments. The literature review also mentions the works of Wash [165], Kauer et al. [87], Furman et al. [58] and others [14, 45, 57, 166].

One of the most noteworthy instances of mental models applied in cybersecurity dates back to 1973, the origin of Public Key Cryptography. Symmetric encryption allows an individual to encrypt and decrypt a message with one key. If two people wish to send messages to one another securely, let's say Alice and Bob for example, then:

1. Alice writes a message and encrypts with with the key K_{AB}.
2. Alice sends this encrypted message to Bob.

3. Bob decrypts the message using the same key K_{AB} and can now read the message.

But how is the key K_{AB} distributed to both Alice and Bob? The key cannot be sent over clear channels because it would be accessible by anyone, rendering the key useless. In 1973, James Ellis, a cryptographer at GCHQ, was tasked with solving this issue. Ellis thought about the concept of a key and padlock, and how locking and unlocking are inverse operations. Ellis' mental model was based on this key and padlock metaphor [48, 49].

1. Alice can buy a key and padlock pair. Alice keeps the key and sends the padlock open to Bob.
2. Bob can write a message and close the padlock, locking the message. This locked message is sent back to Alice.
3. Alice can access the message by opening the padlock by unlocking it with her key.
4. Alice's key is only accessible by her, where as open padlocks are available to anyone who wants to talk to Alice. No keys are exchanged.

Ellis never arrived at a mathematical solution, but he had an intuitive sense of how the process should work, his mental model. In 1973, Clifford Cox realised Ellis' mental model by formulating the trap door one-way function, a function similar in design to RSA. It is important to note that one of the most influential cryptosystem was based on a mental model. Ellis' mental model approach towards solving the problem of Public Key Cryptography serves as an inspiration for Chapter 4: *Developer Mental Models of Public Key Cryptography*.

Open Research Questions:

- When put in the position of Ellis, tasked with designing a security protocol that achieves the conditions of Public Key Cryptography, what are the mental models held by developers and how do these compare to Ellis?

1.1.2 Usability Issues of Security APIs

Studies focused on cryptographic misuse serve as a cornerstone for any attempts made towards improving the usability of cryptographic APIs. Egele et al. [46] conducted an empirical study of cryptographic misuse in Android applications. They found that, of the 11,748 applications inspected, 10,327 (88%) commit at least 1 mistake when compared to basic cryptographic standards. To analyse the Android applications, Egele et al. develop CryptoLint, an application based on Androguard. Androguard disassembles an the code of an application into classes, methods, and the basic building blocks. CryptoLint builds on this representation and further analyses it. CryptoLint evaluates Android applications against 6 rules of cryptography. For example, 'Rule 1: Do not use ECB mode for encryption' [46]. There are other tools, like CryptoLint, that have been developed for the sole purpose of studying the developer's challenges and pinpointing the areas of cryptography where mistakes occur the most. Stransky's Developer Observatory [154] is an online laboratory designed to help security researchers conduct controlled programming experiments while retaining most of the observational aspects of lab studies. It has been used to compare the usability of cryptographic APIs by Acar et al.. Acar et al. studied how the design and noted usability of Python based cryptographic APIs affected the developer's ability to write secure code. This was the same study that found that, for 20% of the tasks that were functionally correct, participants believed that the implementations were also secure, when in reality they were not. The study also found that APIs supported by comprehensive documentation paired with examples of code resulted in an increase of functional code, however this did not extend the security of this code [1, 2].

Nadi et al. [109] performed an empirical study into the obstacles software developers face while using Java cryptographic APIs. They analysed 100 StackOverflow posts, 100 GitHub repositories, and a survey of 48 developers. They found that 57% of the posts were from relatively knowledgeable developers who found the API to be too complex, low-level, leading to difficulties in invoking methods and defining parameters. The survey concluded that the main obstacles that developers face are: a lack of high-level APIs, poor documentation, and badly designed APIs, more specifically, misleading defaults and challenges when debugging. The participants said that they would benefit from task-based solutions through API design, example code shown through documentation, or even as a result of analysis or code generation tools. The literature surrounding cryptographic misuse and understanding the challenges of the developer is well established [1, 36, 51, 65, 74, 91, 92, 107, 111, 146]. This area of research results in three primary forms of advancement:

1. *Recommendations*: Recommendations to improve the usability of cryptographic APIs through design while maintaining a strong level of security.

   Security researchers present recommendations as a response to the challenges developers face while they use cryptographic APIs. Green & Smith [67], for example, presented 10 principles for creating usable, secure cryptographic APIs. The golden rule, as stated by Green & Smith [67], should be the integration of cryptographic functionality into standard APIs so that developers do not have to interact with the cryptographic APIs in the first place.

   Security researchers like Gutmann and Bernstein, looked to improve on the poor state of usability of OpenSSL by designing new cryptographic APIs that would be everything OpenSSL is not: a usable cryptographic API with a high-level abstraction design that limits exposure to low-level cryptographic primitives and offers a set of default-secure algorithms to accommodate the everyday software developer. This cryptographic library should have better documentation, better examples of code, all with the intention of being easier to use compared to OpenSSL. Gutmann documented the design of the Cryptlib library through a trilogy of publications, within which he offered his own set of recommendations for designing a cryptographic API [72–74]. Gutmann followed the recommendation of Saltzer & Schroeder [140] to ensure that Cryptlib would be usable and secure. Later, Bernstein released NaCl, another cryptographic library with claims of being 'easy-to-use' [15]. Bernstein's NaCl [15] builds upon the progress made by Gutmann's Cryptlib [72] with the introduction of the 'crypto-box' API, a high-level abstraction of the encryption process.

Many longstanding recommendations have been based on experience and identifying a need for them at the time they were made. For example, in 2001, Bloch explained how the literature at the time, for Java development, focused on grammar and vocabulary but there was very little guidance on usage. This lead to Effective Java, a book written by Bloch, that offers guidance that focuses primarily on challenges with usage [16]. Later on, in 2006 Bloch adapts his work to address questions like 'How to design a good API?' [17]. Bloch's works was adapted by Green & Smith, tailoring general recommendations for good API design to more specific recommendations for designing usable cryptographic APIs [67].

Recommendations can have a great impact on the design of a cryptographic API, so it is important to understand what these recommendations focus on and what types of challenges they address. This is an area that has not been reviewed in the usable security research community.

Usability is defined through the challenges the developers face while they try to use security APIs. Earlier works identified these challenges through their own experiences with software development [16, 59, 112] or through the poor usability of existing cryptographic APIs [15, 72]. The more recent works have begun to take the approach of studying developers themselves [1, 109].

Open Research Questions:

- What do current recommendations focus on and what types of challenges do they address?
- How, and to what extent, have various recommendations been validated?
- What types of challenges do developers face while they use cryptographic APIs?
- To what extent do principles, aimed at improving the usability of cryptographic APIs [67], address the challenges faced by developers?
- To what extent does the design of cryptographic APIs conform to such principles [67]?

2. *API Design*: OpenSSL is seen as a difficult cryptographic library to use [15, 72, 85]. Bernstein makes the argument that OpenSSL's poor usability can be attributed to the low-level cryptographic API design of the library. With a low-level abstraction, developers are exposed to a wide array of cryptographic primitives and functionalities to choose from for their applications. As discussed earlier, while a cryptographic expert may be familiar with the strengths of different cryptographic primitives to make an educated security decision about which ones to use, an 'every-day' software developer may not be as knowledgeable enough to make these secure decisions. API designers introduce high-level abstractions into cryptographic API design with the purpose of improving usability by preventing the developer from misusing cryptographic primitives and functions of the cryptographic API. For example, NaCl is a high-level cryptographic API developed as a response to reports of the poor usability of OpenSSL. Since then, newer cryptographic libraries, like Libsodium, have been developed as forks to support newer features and applications.

Implementing abstraction into the design of cryptographic APIs introduces the trade-off discussed by LoIacono et al. between *security primitives* and *security controls* [82]. A high-level cryptographic API, although secure, maybe considered rigid and restrictive to software developers whereas a low-level cryptographic API may be considered more flexible allowing developers to make their own choices about the types of cryptographic primitives they want to use. This low-level design could lead to poor security decisions being made by developers who don't know the difference between strong and weak cryptographic primitives.

As discussed earlier, the abstraction seen through the design of the cryptographic API is the representation of the API designer's mental model. The API designer takes a cryptographic concept and represents it through their cryptographic API based on their understanding of the cryptographic concept and their need to improve usability through a high-level API design using abstraction. The software developer also has their own mental model of how a cryptographic concept works.

Open Research Question:

- How do the developers' mental models compare to low- and high- cryptographic API implementations, such as OpenSSL and Libsodium, of Public Key Cryptography?

3. *Tools*: Tools designed to assist developers in writing more secure code. These tools come in the form of static analysers, code generators, or note editors. Cognicrypt [91] is both a code generator and a static analyser created to help developers in implementing secure applications. Cognicrypt generates code for a number of cryptographic tasks such as data encryption, secure channel implementation, and long-term archiving. Cognicrypt [91] also runs a static analyser as the developer is coding, to ensure that the generated code seamlessly integrates with the developer's existing workspace. The static analyser, itself is programmed to follow a series of security rules defined using CrySL [92]. Kruger et al. conducts a similar study to Egele et al., by conducting an empirical evaluation of 10,000 Android applications, finding that 95% of the applications fall under violation of at least 1 cryptographic rule. CrySL served as a means to conduct this analysis, similar to CryptoLint [46, 92].

These are important milestones in the field of usable security. These advancements have addressed cryptographic misuse to a large extent. However, this thesis digs deeper to identify and address the root of the problem. The solutions, so far, are based on the suggestions developers offer, to ease their experience with cryptographic APIs, and the analysis of the challenges they face by the researchers. We ask why they face those challenges in the first place. We argue that this is due to a misalignment between the mental model of the developer and the mental model of the API designer.

1.2 Thesis Objectives

The central research hypothesis that this thesis is built upon is that many errors are due to the misalignments between the developer's mental model and the functionality offered through cryptographic APIs based on the API designer's mental model of standard cryptographic processes. We explore these misalignments through qualitative studies eliciting mental models of developers while they engage with designing cryptographic protocols. We take a look at the guidance that has been proposed by the usable security research community for improving the usability of security APIs through design. This takes the form of a systematic literature review to find out what challenges the current state of literature focuses on and how they came to be. We also study how well these recommendations address the challenges raised by developers themselves. To understand the challenges developers face, while they use cryptographic APIs, we perform a thematic analysis of questions and responses raised by developers on StackOverflow with the aim of identifying *usability issues*. Below we present a series of overarching yet specific objectives based on the open research questions we have raised so far:

1.2.1 What guidance is available for designing security APIs and what is its empirical foundation?

Strategies ranging from Gamma et al.'s design patterns [59] to the design principles of Saltzer & Schroeder [140] have been influential on software engineering practices as well as security API design. We investigate how such strategies may have informed recommendations for designing Application Programming Interfaces (APIs), especially those APIs that provide security and cryptographic functionality.

Over the last 10 years, to help API designers produce security APIs that are more usable, various papers have proposed *usability guidelines, principles and recommendations* [67, 99, 105, 126]—hereafter *recommendations*.

Tracing the ancestry of papers offers a means to systematise 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 [156], no work, to date, has systematised knowledge across this guidance, traced ancestral relationships, as well as the impact of such ancestry on current recommendations for security API usability.

1.2.2 What do software developers struggle with when using cryptographic APIs?

Cryptographic APIs are hard to use. Other works have developed recommendations, guidelines and principles for how to make them more usable—but how can we tell when such usability recommendations, guidelines and principles are not being implemented? To answer this, we need to understand the challenges developers face, when using cryptographic APIs, from the developers themselves, so that we can see which of these usability challenges are addressed by usable security principles [67]. We can also learn which cryptographic libraries do not conform the usable security principles and pinpoint the areas in which they can improve to in turn improve their overall usability. This work would also serves as a means to validate a set of usable security principles [67] that has been very influential in the literature surrounding recommendations for the improvement of security API design.

1.2.3 What are the mental models of the software developers who use these cryptographic APIs?

Public Key Cryptography is a core component of many applications and security networks in use today. Furthermore, Public Key Cryptography consists of many individual cryptographic concepts such as encryption & decryption, hashing, and the use of key-pairs. The discovery of Public Key Cryptography and the mental models used at the time are very well documented, which allows us to compare the mental models we elicit as part of our work with the mental models used back then. The process of Public Key Cryptography can be broken down to address *confidentiality*, *authentication*, and *integrity*, concepts that serve as a form of abstraction, allowing us to present the analysis more clearly as we can analyse the developers' perceptions by these three concepts. Public Key Cryptography is a protocol in itself, proving to be more of a task for developers than simply asking them what they think Public Key Cryptography means. The steps of Public Key Cryptography serve as a way to monitor the developers' proposed design and pinpoint exactly where they deviate from the correct path, if they deviate at all.

Pursuing this objective also allows us to explore the considerations and thoughts that developers have before making decisions about their design. Signs of the developer have an 'attacker mindset' while designing a security protocol can be captured and analysed to see how this mindset impacts the developers' mitigation strategy and how this is represented through their mental models.

Taking the approach of putting software developers in the position of Ellis and tasking them with designing a security protocol that achieves the conditions of Public Key Cryptography; *confidentiality*, *authentication*, *integrity*, without mentioning the term Public Key Cryptography to them, with the aim of eliciting their mental models about how the cryptographic concept works would be a first in the field of usable security. Other studies, such as Wu et al. [168] have performed studies around concepts like encryption, where they have interviewed users about what they think the term encryption mean. The framing of the task, being to design a security protocol to ensure secure communication between two parties, allows us to elicit a richer understanding of the developers mental models and the behaviors by which they are impacted.

1.2.4 What are the misalignments between these mental models and correct usage of the API (in other words, the mental model assumed by the designer of the API)?

API designers design their cryptographic APIs based on their mental models of the standard cryptographic concept. OpenSSL offers a command line interface through which Public Key Cryptography can be achieved but it also offers an envelope interface, a high-level cryptographic API. On the other hand, Lisbodium, offers the crypto_box API another high-level cryptographic API. However, the API provided by both of these libraries are designed by API designers who are cryptography experts and understand the standard Public Key Cryptography concept. So, when trying to find out why software developers struggle to use cryptographic APIs we can make two statements:

- Either the developers' mental model misaligns with the standard way in which Public Key Cryptography is performed. Which means they already have misconceptions about the cryptographic concept even before they start using the cryptographic API.
- Or the developers' mental model is perfectly inline with the standard way in which public key cryptography works but they struggle to understand the API designer's mental model seen through their cryptographic API.

Both statements would greatly help the usable security research community understand the root cause of the problem and begin address the findings through future works.

1.2.5 Can the developers' mental models be used to design usable abstractions for secure programming?

Once the mental models are elicited from the developers, we assess whether or not they can be used for cryptographic API design to help developers with secure programming, as cryptographic APIs are the primary way through which developers perform security tasks. This assessment can lead to three scenarios:

- *Scenario 1:* The developers' mental models are inline with the concept of Public Key Cryptography and with the API designers' mental model, seen through their API design. Here, we can argue that the cryptographic APIs are either usable or can be improved with some small changes based on the developers' mental model. These changes can take the form of an improved clarity of documentation, or some additional resources such as examples of code.
- *Scenario 2:* The developers' mental models are inline with the concept of Public Key Cryptography but misalign with the API designer's mental model. Here, we can make the argument for new usable abstractions in the form of new cryptographic APIs based on commonalities seen through the developers' mental models.
- *Scenario 3:* The developers' mental models deviate far too much from the concept of Public Key Cryptography. Here, we can conclude that the developers' mental models cannot be used as usable abstractions for secure programming. However, it is important to note that secure programming begins far before the development stage of the Secure Software Development Lifecycle (SSDLC). We can apply our understanding of the developers' thought process as a *shift-left* initiative within the SSDLC. The insights gained from the developers' thought process can inform the application design stage of the SSDLC, introducing the use need for cryptographic APIs at an earlier stage.

1.3 Structure of Thesis & Novel Contributions

1.3.1 Chapter 1: Introduction

Chapter 1 gives an introduction to the field of usable security and the challenges of secure programming. We introduce important terms such as *cryptographic APIs*, *abstraction*, *mental models*, which will be used frequently during the thesis. The introduction presents the current state of the art and its limitations. Through our thesis objectives, we show how the thesis builds on the advances that have already been made through a series of novel questions, approaches and contributions.

1.3.2 Chapter 2: From Saltzer & Schroeder to 2021...

Chapter 2 present a systematic literature review centered around security API usability recommendations. Security API usability recommendations can play an important role in influencing and defining corporate policy [11, 102, 122, 144, 160]. However, most of these recommendations come from academic research that has not been systematically reviewed and therefore it is unknown what these recommendations focus on and to what extent they have been validated. Our SLR answers these questions and brings awareness to these recommendations so researchers can begin empirically validating these recommendations and developers and corporations can considering adopting them. We identify and analyse 65 papers spanning 47 years, offering a total of 883 recommendations. We undertake a thematic analysis to identify 7 core ways to improve usability of APIs.

The study presented in this chapter is the first to systematically analyse the recommendations that inform the design of security APIs, crossing scientific communities working on security, APIs, and software engineering. The analysis shows 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, including 13 specifically targeted at providing recommendations to API designers on how to create usable and secure APIs, and 883 recommendations found within the papers, we identified 7 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 organisations. The community has made some strides towards validating recommendations, but more must be done within literature for designing security APIs to improve empirical validation.

Coverage is important alongside validation rates. As part of our analysis, we traced the ancestry of recommendations seen in the initial papers that focused on the design of security APIs. We studied how the recommendations came to be and if they were influenced by earlier works, to what extent were these works validated? 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, 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 should address the earlier literature of the field in order 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.

1.3.3 Chapter 3: Usability Smells - An Analysis of Developers' Struggle With Crypto Libraries

In Chapter 3 we perform a thematic analysis on over 2,400 StackOverflow questions and responses from developers looking for help with 7 cryptographic libraries. The motivation behind this study is to understand the challenges developers face while they use cryptographic APIs and so we identify 16 underlying usability issues and map these to Green & Smith's 10 principles for improving the usability of cryptographic APIs, resulting in the discovery of 4 usability smells. The mapping of the 16 usability issues to the 10 principles proposed by Green & Smith serves as an empirical validation and our contribution to the literature surrounding security API usability recommendations reviewed in Chapter 2.

By mapping the 16 usability smells to the 10 usability principles defined by Green & Smith [67], we can suggest how to improve cryptographic libraries and make them more usable for developers. Our study offers evidence to validate parts of Green & Smith's principles, but also highlights issues that were missed. Their usability principles suggest ways to mitigate most of the issues we identify; however issues associated with the *doesn't play well with others* smell (in particular *build* and *performance* issues) suggest the need for an additional principle to help cover these issues.

Our usability smells capture the general challenges developers face when using cryptographic libraries. Not all libraries smell the same, and improvements to *usable* cryptographic libraries appear to be beneficial with fewer usability smells. By identifying these usability smells we can find the challenges faced by developers and help improve usability. Libraries will perhaps always show signs of these usability smells given the challenges of catering for the requirements of a wide and diverse set of developers and applications; but by integrating usability principles we can at least make them less so.

1.3.4 Chapter 4: Developer Mental Models of Public Key Cryptography

Using a combination of the think-aloud protocol and a diagramming exercise we elicit the developer's mental model, while they attempt to solve the conditions achieved by Public Key Cryptography (PKC). We compare their mental models to the standard process of PKC, which originated from Ellis' mental model in 1970. The analysis of the developers' mental models helps to identify any misconceptions they have as we compare their thought process for solving the conditions of PKC to the correct steps of the PKC process itself. This exercise also gives us an insight into the developers' ability to design a cryptographic protocol. The process of designing their protocols is also analysed as it captures nuances in behavior.

1.3.5 Chapter 5: CryptoBridge - A Need For Effective Challenges & Responses During Secure Software Development

In Chapter 5, we discuss the implications of software designed through the Software Development Lifecycle (SDLC) and although *shift-left* initiatives such as the Secure Software Development Lifecycle (SSDLC) aim to integrate security at every stage of the process, we argue that the use of cryptographic libraries is still considered an afterthought. We discuss how the insights from the developers' thought process, captured through our grounded theory in Chapter 4, can be used as a means to introduce the need for cryptographic libraries as earlier in the *Application Design Stage* of the SSDLC and build future abstractions. We present CryptoBridge as a concept that could be a first step towards distilling the features needed for such an approach.

1.3.6 Chapter 6: Conclusion

We conclude the thesis by discussing the impact of this approach on the usable security research community, and outline directions for future work.

1.4 Publications Emerging from this Thesis

All of the work presented in this thesis, unless otherwise indicated, is that of the author. Some of the work presented in this thesis has been previously published in various venues, with occasional support from other authors as described below.

1. The systematic literature review pertaining to focus of security API design recommendations and their empirical foundation, described in Chapter 2, has been published at the Transactions of Software Engineering Methodology (TOSEM) journal. The systematic literature review has also been accepted for presentation at the International Conference on Software Engineering (ICSE 2023).

   [83, 125] N.Patnaik, A.C.Dwyer, J.Hallett, A.Rashid, *SLR: From Saltzer & Schroeder to 2021...47 years of research on the development and validation of security API recommendations*. In Proceedings of the 2022 ACM Transactions of Software Engineering Methodology (TOSEM).

2. The thematic analysis of the challenges raised by developers while they use cryptographic libraries is covered in Chapter 3. This study was published at the Syposium on Usable Privacy and Security (SOUPS).

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

3. During the thesis we wrote a short paper to highlight the challenges developers face while trying to recruit software developers for security studies. This study was published at the first International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES'2022).

   [116] N.Patnaik, J.Hallett, M.Tahaei, A.Rashid, *If you build it, will they come? Developer recruitment for security studies*, First International Workshop on Recruiting Participants for Empirical Software Engineering, (RoPES'2022).

There are other studies that I have worked on as a co-author which I may reference within this thesis as the work is relevant to this thesis.

1. Does the act of writing a specification for a piece of security sensitive code lead to developers producing more secure code? A question that drove an analysis of code written by 138 developers, half of whom were asked to write a specification. The developers were tasked with securely storing a password. We evaluated their code using Naiakshina's guidelines for correctly securing passwords. We found that the group who wrote a specification before the coding exercise produced code that was marginally more secure than the group who did not write a specification. This study was published at the International Conference on Software Engineering in 2021.

   [76] J.Hallett, N.Patnaik, B.Shreeve, and A.Rashid, *"Do this! Do that!, And nothing will happen. Do specifications lead to securely stored passwords?"*. IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp 486-498.

2. We distilled and summarised the categories, challenges, behaviors, and interventions from the results of literature surrounding developer-centered studies. The analysis lead to the identification of 5 misplaced beliefs or *tropes* about developers embedded in the core design of security APIs and tools. This study was published in the Secure Development Conference in 2021.

   [29] P.D.Chowdhury, J.Hallett, N.Patnaik, M.Tahaei, and A.Rashid, *Developers are neither enemies nor users: They are collaborators*, in 2021 IEEE Secure Development Conference (SecDev), IEEE, 2021, pp.47-55.

Chapter 2

From Saltzer & Schroeder to 2021...

Producing secure software is challenging. The poor usability of security 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 systematise knowledge regarding these recommendations. We identify and analyse 65 papers, offering 883 recommendations. Through thematic analysis, we identify 7 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 *organisational software development practices*. By tracing and analysing 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.

2.1 A Systematic Literature Review on the development and validation of Security API recommendations

Programming is hard to do well, and, even more so, securely. Developers frequently combine functions from APIs; but some are notoriously difficult to use correctly [96, 134], with cryptography and security libraries often singled out as being particularly obtuse [85, 109]. Strategies ranging from Gamma et al.'s design patterns [59] to the design principles of Saltzer & Schroeder [140] 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 last 10 years, to help API designers produce security APIs that are more usable, various papers have proposed *usability guidelines, principles and recommendations* [67, 99, 105, 126]—hereafter *recommendations*.

Tracing the ancestry of papers offers a means to systematise 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 [156], no work, to date, has systematised 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 analyse their ancestry, identifying 883 recommendations in 65 papers (categorised in Table 2.1). These include papers offering general API design recommendations, those providing general security best practice, and broader software engineering design guidance and recommendations. We categorise these recommendations and analyse their ancestry to answer 3 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 7 broad categories. While many papers recommend improving documentation to assist developers [12, 16, 17, 105, 112, 123, 126, 134, 160, 169], 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 3 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 2.4). 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 organisational factors,

Table 2.1: Count of Recommendations (Papers) analysed 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, 22, 46, 62, 67, 74, 99, 100, 105, 118, 120, 126, 164]
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)	[12, 16, 17, 30, 41, 70, 79, 89, 96, 115, 123, 134, 135, 156, 169]
Software engineering recommendations	Literature that is around generic software engineering and best practices in the form of recommendations.	207 (17)	[6, 27, 35, 56, 59, 68, 69, 78, 80, 81, 86, 106, 108, 112, 113, 130, 149]
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)	[7, 11, 20, 25, 51, 63, 65, 73, 75, 95, 97, 98, 102, 121, 122, 139, 140, 144, 145, 160]
Total		883 (65)	

and that security-focused papers specifically touching on APIs tend to ignore documentation or validation aspects (RQ1: Section 2.4.1).

- 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 2.4.2).
- A well defined ancestry linking much of the security API guidance going back 47 years to Saltzer & Schroeder's seminal work in 1974 [139], amongst others (RQ3: Section 2.4.3).

Through a systematic literature review, we identify and analyse 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 *8 meta-recommendations* which summarise 47 years of design guidance targeted at software engineering and computer security.

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

2.2 An Introduction to Saltzer & 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 & Schroeder [139].

2.2.1 Saltzer & Schroeder's Design Principles

In 1974, Saltzer addressed the challenge of protection and control of information sharing in Multics (Multiplexed Information and Computing Service). The Multics system served as a case study for protection mechanisms that allowed for a controlled sharing of information between parties in an on-line information storing system. The design of the Multics system aimed to achieve two functional objectives. If the system design is setup in a way where a single administrator is faced with making too many administrative decisions, the systems' performance can become effected, to the point where users may start showing habits of bypassing the administrator, which in turn could possible result in compromising security. To address this challenge the Multics system aimed to introduce decentralisation into its design. In practice this meant that some administrative responsibilities were also placed upon the users of the system, not only the administrator. The second functional objective of the Multics system was to provide the user with a set of functions that allow them to build a protection environment with custom access requests based on their requirements. This functionality was based on the assumption that there may be some users who will require protection schemes that were not anticipated and not integrated into the default standard of the original design. The Multics system allows for any user to build their own protected subsystem, comprised of a collection of programs and data. The protected subsystems allow for the data to be accessed by only the programs in the subsystem. This implementation of a protected subsystem would allow a user to define an access control scheme to their requirements [139].

Saltzer offered 5 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 & Schroeder presented 8 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 used by several users. The key challenge Saltzer & 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 [140]. Saltzer & Schroeder's work became very influential and applicable to a wide range of fields such as enforcing security policies [143], evaluating the Java security architecture [64], and minimising user-related faults in information systems security [148]. However, it was in 1995 when Saltzer & Schroeder's principles were first applied to the design of security APIs through Cryptlib [72]. In 1995, Gutmann designed Cryptlib, a cryptographic library, based on the adaptation of Saltzer & Schroeder's principles [72].

Table 2.2: Saltzer & Schroeder's 8 principles to guide the design of information protection mechanisms in computer systems [140].

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	Minimise 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.

2.3 Method

The methodology behind performing the systematic literature review is broken down and explained in the following 4 sections:

1. Identifying the 13 Security API designer papers.
2. Tracing the ancestry of the 13 Security API designer papers.
3. Identifying Actionable Recommendations.
4. Categorising recommendations.

2.3.1 Identifying the 13 Security API designer papers

2.3.1.1 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:

```
          Usability|Guidelines|
API (Principles|Design|Librar(y|ies))? (Security)?
```

We used a snowball-method [167] on the papers identified from the online search; and using forward and backward snowballing, we checked to see if there were other relevant papers.

2.3.1.2 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 [167] on the papers we found through our review of these ten 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 & Privacy (Oakland),
- IEEE Transactions on Software Engineering (TSE),
- International Conference on Software Engineering (ICSE),
- USENIX Security Symposium (USENIX),
- International Symposium on Usable Privacy & Security (SOUPS),
- ACM Conference on Computer and Communications Security (CCS),
- Network and Distributed System Security Symposium (NDSS),
- ACM Foundations of Software Engineering (FSE), and
- 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 2.3). We identified an initial 45 candidate papers, through step 1 and 2, that provided Security API Designer recommendations.

2.3.1.3 Step 3: Selecting Relevant Work

Each paper was reviewed independently by 3 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 which 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.
- 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 [22]. Brown et al. identified a series of effective checks to find bugs in 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 [22].

Georgiev et al. identified poorly designed SSL implementations vulnerable to MITM attacks [62]. 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.

2.3.2 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 2.6). At the end of this step we had 156 papers including the 13 Security API designer papers (Table 2.1, Figure 2.1).

2.3.3 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 [$\cdots$]. If the API is not self-explanatory or worse gives the false impression that it is, developers will make dangerous mistakes." [67]

Offered Guidance

> "Economy of mechanism: Keep the design as simple and small as possible. [$\cdots$] design and implementation errors that result in unwanted access paths will not be noticed during normal use [$\cdots$]. 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" [140]

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." [17]

From the 156 papers, we identified 65 papers providing 883 actionable recommendations for improving usability and security (Table 2.3).

From our 65 actionable papers, 2 authors allocated each paper to one of four paper types, shown in Table 2.1, based on an inductive process derived from the papers themselves. This results in *Security API designer* and 3 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.

2.3.4 Categorising the Recommendations

To understand the different areas of recommendation focus, we categorised our 65 actionable papers. To alleviate bias from predefined categorisation the analysis followed an inductive, bottom-up approach [133], drawing out recommendation themes.

1. Two authors, in joint discussion, selected 50 recommendations to identify different *codes* in order to build a mutual understanding of the recommendations. An initial *codebook* [5] with 28 codes was created.

2. Over three iterations, the 883 recommendations were categorised 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 7 categories, and 36 descriptive sub-categories, as shown in Table 2.4. 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 [123] 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”

Table 2.3: 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, 7, 22, 67, 98, 99]
USENIX Security	5	[73, 74, 78, 120, 164]
IEEE Software	4	[96, 113, 134, 160]
SOUPS	4	[11, 65, 118, 126]
IEEE VL/HCC	3	[12, 89, 156]
CACM	3	[106, 108, 139]
ACM CCS	3	[46, 51, 62]
IEEE TSE	2	[75, 80]
Carnegie-Mellon University ..	2	[123, 144]
Elsevier VL&C	2	[6, 69]
OWASP	2	[121, 122]
Addison Wesley	2	[16, 56]
IEEE/ACM ICSE	1	[100]
ACM OOPSLA	1	[17]
IEEE QRS	1	[105]
ECOOP	1	[59]
ACM CHI	1	[112]
PPIG	1	[30]
People & Computing	1	[68]
ACM SIGDOC	1	[86]
EclipseCon	1	[41]
ACM CCSC	1	[115]
HCSE	1	[70]
IJCSNS	1	[169]
Microsoft	1	[102]
BSIMM	1	[25]
Proceedings of the IEEE	1	[140]
ACM Software Engineering Notes	1	[97]
IEEE SESS	1	[20]
Secure Software, Inc.	1	[145]
IEEE ACSAC	1	[95]
IEEE COMPSAC	1	[81]
Human-Computer Interaction ..	1	[130]
ACM TOIS	1	[27]
National Computer Conference .	1	[149]
Pearson Education	1	[35]
ACM Queue	1	[79]
ESE	1	[135]
IEEE SCAM	1	[63]
Overall	65	

Table 2.4: 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 & 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.
• Compartmentalisation	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.
• Standardised	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.
Organisational Factors	**How organisations 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 organisation develops knowledge and practice of security.
• Training	The delivering of training for organisations and their members.
• Third Party	How organisations interface with third parties and third party components.
• Regulatory	Any regulatory, legal, or compliance that an organisation does.
• Risk Assessment & 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 towards 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	Offer enough information to effectively communicate and provide understanding.
• Validation of Activity	Providing API users and API developers tools that check their activities.

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 [31]), and using only a single category per recommendation (as Cohen's κ only deals with single categorisations per subject), we calculated a κ of 0.74 when mapping the categories, and 0.76 when mapping the descriptors—indicating substantial agreement [94].

2.4 Findings

Table 2.4 outlines the 7 recommendation categories and 36 descriptor sub-categories. The 7 categories describe overarching themes and topics about which papers make recommendations. 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." [16]

In contrast, the *Economy of Mechanism* descriptor identifies the simple code construction to avoid errors—found in Nino et al.'s *"be minimal"* [115], Grill et al.'s *"Do not provide multiple ways to achieve one thing"* [70], OWASP's *"Keep It Simple, Stupid Principle"* [121] or Saltzer & Schroeder's *Economy of Mechanism* principle [140]; from which we name the descriptor.

We also observe recommendations on how to document code (the *Documentation* category)—typically focused on clear explanation, communication, standardisation 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" [6]

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

Table 2.5 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 5 remaining categories (ranging between 7–14%).

Table 2.5: Recommendations mapped to category and paper type. 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.

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%)

2.4.1 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 *Organisational 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 [67, 126] 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 2.1).

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%). 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 categorised under *Organisational 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 organisations such as Microsoft [102]. 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 [160] and Assal [11] (Figure 2.1). Many of the recommendations in this category come from corporate grey literature (such as Microsoft's SDL [102], The BSIMM framework [25] and OWASP [122]). For example, Microsoft's SDL encourages developers to *"Establish a standard incident response process"* [102] 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 organisations should *"educate executives"* [25] so that *decision-makers* in an organisation are sufficiently knowledgeable about security (*Organisational Factors: Training*). Recommendations focused on organisations 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 organisations and the impact of external contexts on being secure. The recommendations reflect how organisational 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 [62]. 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 [62].

2.4.2 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 4 ancestor-descendant relationships can risk propagating ineffective recommendations.

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 2.6). Within the ancestor–descendant relationships, we make a distinction between empirical validation and the 4 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 2.6. 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* [16] to his 2006 paper *How to Design a Good API and Why it Matters* [17]. 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

Table 2.6: Codebook used for describing 5 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 analyse 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.

usability recommendations—with the latter paper used frequently by Security API designer papers.

Borrowed. Myers and Stylos [108] refer to the ancestry between Grill et al. [70] and Nielsen and Molich [114]:

> "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." [108]

Adaptation. In 2017, Acar et al. [1] evaluated and compared the usability of 5 Python-based cryptographic libraries. To evaluate the usability of these cryptographic libraries, Acar et al. *adapted* recommendations from Bloch [17], and from Green & Smith [67].

> "We adapt guidelines from these various sources [Bloch [17], Green & Smith [67]] to evaluate the APIs we examine." [1]

Comparison. Smith [150] reflects on the work of Saltzer [139] along with the 1975 paper written with Schroeder [140] by *comparing* the *principles* of Saltzer & 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." [150]

Empirical. In 2019, Patnaik et al. [126] *empirically* evaluate the 10 principles designed to improve usability and security of APIs by Green & Smith [67].

> "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." [126]

2.4.2.1 Validation by Paper Type

Table 2.7: Rates of different kinds of paper validation for different categories of guideline papers in the literature. The overall values account for single papers being validated multiple times.

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%)

Table 2.7 summarises 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 2.1; 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 [112] and Saltzer & Schroeder's principles [140]. Though efforts have been made to empirically validate earlier papers [112, 140], 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. Out of 13 Security API designer papers, only 3 papers [46, 62, 67] are empirically validated.

As we create further recommendations, we must consider the role of ancestry, and the validation of what it recommends, in order 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 4 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.

2.4.2.2 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 2.8 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%) (Table 2.7, Table 2.8).

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 2.8). 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 2.8).

Table 2.8: Counts of recommendations that have been empirically validated or part of other ancestor–descendant relationships across the 7 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%)

2.4.3 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 [74] or subsequently through Green & Smith [67].

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

First, *abstract* recommendations such as by Green & Smith [67] apply to a number of tools, applications, and contexts. Second, there are *concrete* recommendations as identified by the ancestries of tools and applications [46, 99]. 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.” [67]

Concrete Recommendation. “Client-side validation must be thoroughly tested for consistency with server-side validation logic. WARDroid can help in identifying potential inconsistencies” [99]

From the examples given above, Green & 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. On the other hand, Mendoza et al. offer a concrete recommendation. The recommendation is a policy expressed through WarDroid [99] 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 2.1). This shows instances of empirical validation and the ancestor–descendant relationships between papers (and 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 & Schroeder [140], Bloch [16, 17], and Gamma et al. [59]. 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 & Schroeder are directly referenced by security API design works of 2020.

2.4.3.1 Saltzer & Schroeder: Once Upon A Time

In 1975, Saltzer & Schroeder presented 8 design principles to help guide the design of protection mechanisms and prevent security flaws [140]. The design principles were *adapted* by a revision of material originally published by Saltzer in 1974 [139].

Saltzer & Schroeder’s work has been thoroughly influential through many relationships by works very different from each other, addressing fields from security policies [143] from security for Java applications [64] to cryptographic APIs. Their work has also been *empirically* validated [148]. This level of influence establishes Salzter & 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 & Schroeder and 7 of our 13 security API designer papers. Gutmann *adapted* Saltzer & Schroeder's design principles when designing Cryptlib [72].

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." [72]

Bernstein et al. built on Cryptlib and presented NaCl, an even more abstracted cryptographic API that *compared* NaCl to Cryptlib in detail [15]. NaCl, itself, has forks such as Libsodium which also has forks including Monocypher. We can see how strongly Gutmann's *adaptation* of Saltzer & 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 [73]. 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 [74]. These recommendations are the oldest of the security API designer papers. Compared to Saltzer & Schroeder, Gutmann's recommendations [74] have not been as widely validated by or related to other works.

The Emergence of Green & Smith's Principles: In 2016, Green & Smith presented 10 recommendations to help developers create more usable and secure cryptographic APIs [67]. 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 [17]. Green & Smith's work is the ancestor, in the ancestor–descendant relationship, to 4 security API designer papers [1, 105, 118, 164]. Their recommendations are also empirically validated by Patnaik et al. [126], another security API designer paper.

Patnaik et al. [126] offered an *empirical* validation in this chain through evaluating the 10 Green & Smith [67] principles. Through an analysis of over 2,400 Stack Overflow questions and responses from developers facing challenges using 7 cryptographic libraries, they found 16 usability issues which were mapped against the 10 principles of Green & Smith [67]. 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 & Smith's principles. Based on this, they derived additional recommendations: *4 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 [105]. 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 & Smith:

> "Compared to Green & Smith's top ten principles, our recommendations are more specific but do not conflict with their suggestions." [105]

Acar et al. [1] *adapted* Green & 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. [118] *distill* the work of Green & 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. analysed the developer's personal traits, such as; perception of correctness, familiarity with code, and level of experience.

Votipka et al. *adapted* Green & Smith to help understand what errors developers tended to make and why. Votipka et al. analysed 94 submissions of code attempting security problems, resulting in 182 identified unique security vulnerabilities [164].

Votipka et al.'s recommendations from 2020 is the latest in an ancestral chain dating back to Saltzer & Schroeder's design principles of 1975, aimed at protection mechanisms [140]. Authors like Gutmann and Green & Smith played a pivotal role when tailoring Saltzer & Schroeder's design principles towards the security API design recommendations of today [67, 74]. However, it is interesting to identify a direct and contemporary relationship between Votipka et al. and Saltzer & Schroeder. Votipka et al. *borrowed* Saltzer & 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 & Schroeder's principles are still relevant today and flexible enough to address the challenges of designing security APIs directly.

2.4.3.2 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 [8, 66]. 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 [16]. Throughout the book, Bloch evaluates and compares his recommendations to Gamma et al.'s design patterns [59].

"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]" [16]

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 [17]. These recommendations were later adapted in 2016 by Green & Smith to improve the usability of security APIs through design [67].

Bloch's work is also *adapted* by Acar et al., who also *adapts* the works of many, including Green & Smith [67], Henning et al. [79], and Nielsen [112], 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 & Smith [16, 17], and those that Green & Smith inform [1, 105, 118, 126, 164]. In Bloch's ancestry, we do not find any explicit evidence that Bloch's recommendations have been empirically validated as they moved into Green & Smith, though there is some traceability to Gamma et al.'s design patterns—that are rooted in observations of developers' problem-solving practices. Bloch [16] discusses the many architectural advantages of Gamma et al.'s design patterns, and more specifically Gamma et al.'s factory pattern [59]. This suggests that we need not only further empirical validation of Green & Smith, and the ancestry this work builds on.

2.4.3.3 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 [62]. 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 [62].

Georgiev et al.'s recommendations are *empirically* validated by O'Neill et al. [120], another security API design paper, who also *empirically* validate the works of Brubaker et al. [23] and Fahl et al. [51]. 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 [120]. 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 [119]. 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. analyse 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 [100]. 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 [62].

> "*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 analysing code." [100]

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" [46]. Egele et al. *adapt* the work of Bellare et al. [13] and Desnos' Androguard [42].

> "Our tool, called CryptoLint, is based upon the Androguard Android program analysis framework. [⋯] We adopt the notation used by Bellare and Rogaway." [46]

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 4 security API designer papers [46, 62, 100, 120], we can see that a good foundation is forming. In particular, Georgiev et al.'s [62] recommendations have been *empirically* validated by O'Neill et al. [120], and *compared* by Meng et al. [100], Brubaker et al. [23], and Fahl et al. [51]. 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.

2.5 Threats to Validity

We identify four main threats to validity.

First, our search terms on Google Scholar, IEEExplore and ACM DL (see Section 2.3) may have overlooked some papers relevant to our study. We believe we have mitigated this threat and are confident

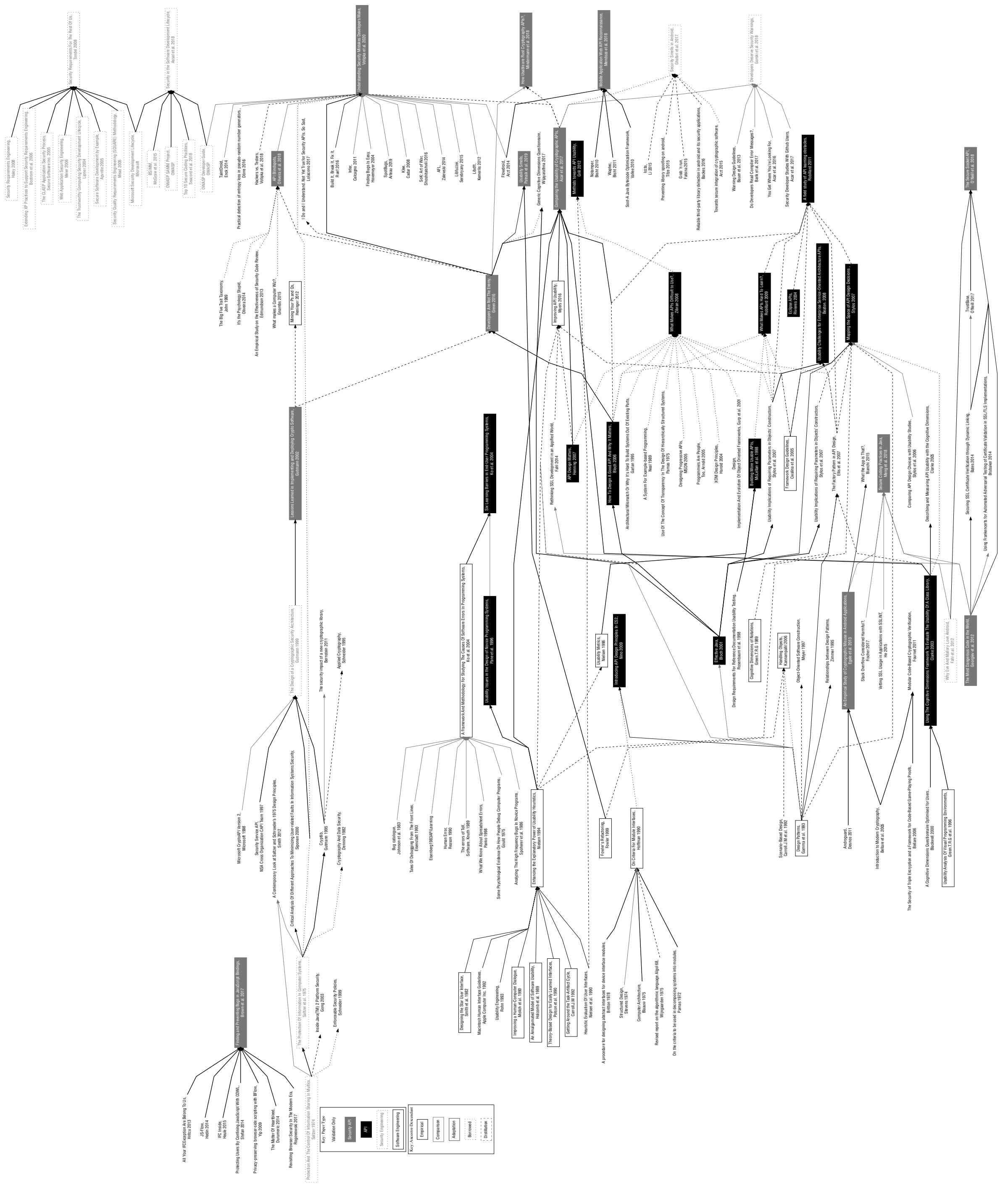


Figure 2.1: Graph showing links between papers where one paper has built upon the work of another, and key. These were identified through the paper survey, using Security API designer papers with recommendations to identify and validate how knowledge has been translated.

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 2.3 lists all venues and publishers for our 65 papers providing actionable recommendations.

Second, the categorisation was conducted inductively. In order to ensure the categorisation was consistent we did an independent coding and calculated Cohen's κ [31], demonstrating that our categorisation was consistent between coders. However, roughly a fifth of recommendations have two categories. Cohen's κ is not designed for data with multiple categories, so when calculating inter-rater reliability we used only the first categorisation. 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 & Smith [67] against 16 usability issues faced by software developers when trying to use 7 cryptographic libraries [126].

In 2002, Gutmann studies his own design principles from 1999, on which Cryptlib was designed [73]. As a result of studying his own design principles, Gutmann presented a set of new recommendations in the form of lessons learned from the process of designing a cryptographic library [74].

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 & Smith [67, 105].

'Compared to Green & Smith's top ten principles, our recommendations are more specific but do not conflict with their recommendations' [105].

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 7 categories and 36 sub-categories in Table 4. We also perform an analysis using this data in Table 5. The information shown through Table 2.4 and Table 2.5 identifies what challenges the 883 recommendations focus on.

2.6 Discussion

2.6.1 The Importance of the Longstanding Principles

What makes the works of Saltzer & 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 [16, 17, 59, 112, 140]. This is no clearer than in Saltzer & Schroeder's case [140] where they define a set of recommendations to address the design challenges of protecting information, stored on computers, from unauthorised access. Gutmann facilitated the transition from security engineering to security API design by using Saltzer & Schroeder's recommendations to design the Cryptlib cryptographic API [72]. Gutmann's work is later related to by Green & Smith, from which many other Security API designer papers grew [67]. This evolution was possible primarily because Saltzer & Schroeder's recommendations were actionable. Saltzer & Schroeder's recommendations are adapted again in 2020 directly by Votipka et al. [164], 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 [16]. In 2006, Bloch adapts his own work towards designing good APIs [17]. Green & Smith tailor Bloch's API design recommendations for security API designers [67]. The wide-spread influence seen through Gamma et al.'s ancestry explains why it referred to through many ancestor-descendant relationships.

Saltzer & 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 [112]. 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 [12, 16, 17, 41, 105, 112, 123, 126, 134, 160, 169].

Before going on to advise on how one can ease novice programmers into programming, Pane and Myers [123] 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" [112]

Pane and Myers go on to inspire others and bring usability specifically to developers. 20 years later Green & Smith describe their 10 principles for creating usable and secure crypto APIs [67]. For example, "Make APIs easy to use, even without documentation" [67]. Yet again, these earlier usability guidelines have been restated, rediscovered and then returned to.

11 recommendations say specifically to use (and occasionally not to use [70]) Gamma et al.'s *Design Patterns* and 6 reference *Factory Patterns* directly. 4 papers related to Gamma et al.'s work, and a further 2 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 specialised 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.

2.6.2 More Validation Please!

How did Saltzer & Schroeder's work from 1975 remain relevant over the last 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 & Schroeder, would Votipka [164] 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 towards 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 2.7).

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 in order 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. In order 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 analysed.

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. [62]. 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.

2.6.3 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* (Table 2.4, Table 2.5). 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 2.1. If empirical validation has been performed, 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 haven't 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 2.1, 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 the time focused on grammar and vocabulary but found there to be a lack of guidance surrounding correct usage when programming in Java [16]. Later in 2006, Bloch adapts this work to address the challenges of designing a good API [17]. 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 [67, 73, 74].*** Green & Smith's recommendations, to help improve the usability of security APIs, have been empirically validated by Patnaik et al. [126]. Green & Smith's work has also influenced 5 of the 13 Security API designer papers through the 4 ancestor-descendant relationships [1, 105, 118, 126, 164]. 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 [72–74]. Gutmann's work has also been *adapted* by Green & Smith and introduced to the field of designing security APIs.

2. ***Security API designer papers in need of validation [1, 22, 99, 100, 118, 120, 126, 164].*** These papers are newer compared to Green & 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 & 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 & Schroeder, Gutmann, and Green & Smith were empirically validated, resulting in many works citing and forming ancestor-descendant relationships with these three papers and progressing their recommendations through time [67, 73, 140]. 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 [25, 102, 121].*** Many of the works categorised as *security engineering papers* are published by corporations such as Microsoft [102]. 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 2.4). *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 [16, 17, 59, 74, 112, 140].*** Researchers and engineers can benefit from revisiting these works to applying their recommendations to the designing challenges of today. Works such as Saltzer & Schroeder [140] have proved to still be relevant and actionable through Votipka et al. [164].

2.6.4 Meta-Recommendations

Our categorisation 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 analysing 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 [11, 12, 16, 17, 25, 102, 115, 122, 144]. 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 [17, 41, 59, 84, 115, 139–141]. 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 [25, 95, 140, 145]. Relevance: The Developers.*** Design security in from the start by compartmentalising components, and having sensible defaults. Have a plan for dealing with bugs and defects.

4. ***Show, and Tell [17, 70, 105, 126, 160]. Relevance: The Developers.*** Documentation matters! Document how the APIs work. Document how programmers should use them. Provide exemplars. Standardise as much as possible. Make sure the documentation is easy to find and read.

5. ***API Developers are not an Island [25, 102, 122, 145]. Relevance: The Company and The Research Community.*** An API might be for programmers to use, but they are often maintained and

managed within organisations. Executives need training to make good decisions, and organisations 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 [11, 17, 25, 75, 97, 98, 102, 144, 147]. 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 [12, 30, 67–70, 81, 88, 89, 106, 112, 113, 123, 156, 169]. 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 isn't just for users.

These 7 guiding principles summarise our 7 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 analysing 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 [73].

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 out 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: [17, 59, 112, 140] 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 & Schroeder, Bloch, and Gamma et al., suggest they are still helpful and relevant today.

2.7 Conclusion

Our study is the first to systematically analyse the recommendations that inform Security API designer papers, crossing scientific communities working on security, APIs, and software engineering. Our research questions systematise 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 on how to create usable and secure APIs, and 883 recommendations found within the papers, we identified 7 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 organisations. The community has made some strides towards 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, 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 should address the earlier literature of the usability field in order 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 don't need to reinvent the wheel so much as assess and renovate the parts to make them roadworthy for usable Security API designer recommendations today.

The recommendations found through our systematic literature review reflect the research communities' perception of the challenges developers' face and how to address them through improved security API design. Only a small number of works, like Acar et al. [1] have actually based their recommendations off their observations of developers using security APIs. Other, well cited, works such as Bloch [16, 17] have presented recommendations based off their experience, which were later adapted by Green & Smith, who proposed 10 principles for improving the usability of security APIs through design. As we found, Green & Smith's work served as a bridge between literature focused on API design, and software engineering and 5 of the 13 Security API designer papers [1, 105, 118, 126, 164]. But how can we be sure that the challenges against which Green & Smith proposed 10 principles are the challenges actually faced by developers? In the next chapter, we present an empirical study that analyses and identify the challenges faced by developers and map these challenges to the 10 principles of Green & Smith to learn to what extent do the principles address the challenges and if there are any types of challenges that do not fall under the umbrella of Green & Smith's 10 principles.

CHAPTER 3

USABILITY SMELLS

Green & Smith propose ten principles to make cryptography libraries more usable [67], but to what extent do the libraries implement these principles? We undertook a thematic analysis of over 2,400 questions and responses from developers seeking help with 7 cryptography libraries on Stack Overflow; analysing them to identify 16 underlying usability issues and studying see how prevalent they were across the 3 cryptography libraries for which we had the most questions for on Stack Overflow. Mapping our usability issues to Green & Smith's usability principles we identify 4 *usability smells* where the principles are not being observed. We suggest what developers may struggle the most with in the cryptography libraries, and where significant usability gains may be had for developers working to make libraries more usable.

3.1 An Analysis of Developers' Struggle With Cryptographic Libraries

Cryptographic APIs are hard to use. Other works have developed recommendations, guidelines and principles for how to make them more usable—but how can we tell when such usability recommendations, guidelines and principles are not being implemented? In this paper we focus on the ten principles proposed by Green & Smith [67] (reproduced in Figure 3.1). We investigate two key questions: (i) what are the issues that developers face when using seven cryptography libraries and (ii) what are the telltale signs that one of the 10 usability principles is being violated?

Code smells are indicators that a piece of software code may be of lower quality than desired [55]. A code smell signifies that, while a piece of code may not be broken, it is violating a design principle and may be fragile and prone to failure. For example, Fowler defines the *Shotgun Surgery* smell as:

> "You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard

to find, and it's easy to miss an important change." [55]

Code that smells of shotgun surgery may be correct and pass all the tests, but the smell suggests that there may be a deeper issue with the code's structure.

Following the idea of a code smell, a *usability smell* is an indicator that an interface may be difficult to use for its intended users. Past work has focused on usability smells in graphical user interfaces (GUIs)—indicators that end users may struggle to use an application [3, 77]. However, usability issues are not limited to GUIs. Developers struggle with programming interfaces in the same way that users struggle with user interfaces. For example, past work has suggested that improving the quality of documentation would lead to developers needing to ask fewer questions about how to use libraries [105]. If the developer is unfamiliar with the library they will rely on the documentation provided with the library and their own programming knowledge to implement the required cryptographic tasks. If we look at a developer question and answer site, such as Stack Overflow (a popular developer question and answer help website), we might expect to see fewer questions asking for help with the basic usage of a library if it has improved its API documentation. The idea of improving API documentation and introducing task-based examples for improving the overall usability of cryptographic libraries is well supported [109, 135, 161]. However, as our analysis shows, these smells are present across the cryptography libraries we examined and all can make usability improvements to help developers use them successfully.

In order to identify developers' struggles with cryptographic libraries, we analyse 2,491 Stack Overflow questions. We examine questions about seven cryptographic libraries (Table 3.2), selected for their popularity and to encompass a broad range of languages and use-cases. We conduct a thematic analysis [21, 34, 138] of the questions and answers looking for the underlying reason the question was asked—be that because of missing documentation, confusion around an API, lack of cryptographic knowledge, or developers preferring Stack Overflow to other resources. We identify 16 thematic issues across our corpus of questions and measure their prevalence across the different libraries. We relate these issues back to Green & Smith's usability principles and identify *four usability smells* that indicate that *specific* principles are not being implemented fully. Finally we make suggestions, based on the prevalence of smells in each of the libraries, as to how library developers can better implement the principles to reduce the smells and make their API more usable.

The novel contributions of our investigation are as follows:

- An empirical validation of Green & Smith's principles showing when a principle is not being applied but also identifying issues that Green & Smith's principles currently do not capture.
- The thematic analysis of 2,491 Stack Overflow questions to assess the usability of cryptographic libraries.
- Identification of 16 thematic issues across 7 cryptographic libraries—capturing developers' struggles with regards to the usability of these libraries codified into four usability smells (Needs a

super sleuth, Confusion reigns, Needs a post mortem, and Doesn't play well with others) which are signs that particular Green & Smith principles are not being fully implemented by a given library; before giving an overview of the prevalence of theses 16 issues, and 4 smells in 3 of the libraries (those that had over a hundred questions with a score ≥ 2).

Table 3.1: Green & Smith's 10 usable cryptography API principles, reproduced from [67]. We have given each principle a short name to allow easy reference throughout the thesis.

Principle	Description
Abstract	Integrate cryptographic functionality into standard APIs so regular developers do not have to interact with cryptographic APIs in the first place.
Powerful	Sufficiently powerful to satisfy both security and non-security requirements.
Comprehensible	Easy to learn, even without cryptographic expertise.
Ergonomic	Don't break the developer's paradigm.
Intuitive	Easy to use, even without documentation.
Failing	Hard to misuse. Incorrect use should lead to visible errors.
Safe	Defaults should be safe and never ambiguous.
Testable	Testing mode. If developers need to run tests they can reduce the security for convenience.
Readable	Easy to read and maintain code that uses it/Updatability.
Explained	Assist with/handle end-user interaction, and provide error messages where possible.

Library	URL
OpenSSL	`https://github.com/openssl/openssl`
NaCl	`http://nacl.cr.yp.to`
libsodium	`https://github.com/jedisct1/libsodium`
Bouncy Castle	`https://bouncycastle.org/java.html`
SJCL	`https://github.com/bitwiseshiftleft/sjcl`
Crypto-JS	`https://github.com/brix/crypto-js`
PyCrypto	`https://www.dlitz.net/software/pycrypto/`

Table 3.2: Cryptography libraries examined in this paper.

3.2 Background and Related Work

The background and related work falls into two broad categories: research on usability issues of APIs in general; and work focusing on such issues in cryptography and security libraries.

3.2.1 Usability issues of APIs

Many studies have addressed the usability of APIs and why they can be difficult to learn and use. Zibran et al. [170] reviewed 1,513 bug posts across five different repositories to identify the API usability issues that were reflected in the bug posts by the developers who used the APIs. They found 22 different API usability factors. We adopt a similar approach and review the questions developers have about each one of our selected cryptographic libraries and see what prevalent usability issues arise. However, we investigate further to identify the usability smells from each library that contribute to the violation of the Green & Smith principles.

Other work has explored ways to measure the usability of existing APIs. Scheller & Kúhn proposed a framework for measuring an API's usability against a set of usability aspects [142]. Dekel & Herbsleb noted that many APIs place notes about when it is appropriate to use certain functions [39]. They developed a tool (eMoose) to integrate these notes into developer's editors (when using an annotated API) and found that developers who used their tool debugged programs quicker than those who only had access to the documentation. In follow up work [40] they noted that the key behind eMoose's success was not that eMoose made the notes immediately available, but rather that it helped provide a *scent* for programmers trying to debug their code—a hint that there was something that *could* go wrong and that prompted them to further read the documentation. Our work identifies *usability smells* that library developers and maintainers can use to understand how they may improve the usability of their libraries in line with the Green & Smith principles.

Helping developers avoid mistakes has been studied extensively with many papers suggesting ways APIs can be improved to help avoid mistakes and to speed up debugging when they inevitably occur. Bloch asserted general principles for API design that would produce a usable API [17]. These principles were summarised as 39 different maxims, though Bloch noted that good API design is a craft and couldn't be entirely captured by lists of rules. Others have proposed similar lists of metrics, often developed from studying or surveying developers. Ko & Yann studied the way developers use APIs [90]. They found that developers do not only need detailed worked examples but also good explanations of the concepts, parameters and ideas behind the API's design. Piccioni et al. [129] ran a study to assess the usability of an API by comparing the programmer's expectations to their performance. They found that issues with naming convention and types confused programmers, poor documentation made programming harder and that overly flexible APIs confused less experienced programmers. Clarke & Becker adapted the *cognitive dimensions* framework, used to describe the usability of user interfaces [69], to evaluate the usability of a class' API [30]. They suggested a list of ten dimensions that APIs should be judged on based on the original cognitive dimensions framework and two new ones that capture how much work any individual operation does.

As developers have changed so have their sources of documentation. Stack Overflow is increasingly used as a primary source of documentation. Parnin et al. surveyed questions about three popular (non-cryptographic) APIs on the site. They found that, on average, 80% of the API functions would be covered by at least one Stack Overflow question, but that only a relatively small pool of *experts* answered them [124]. Treude & Robillard looked at ways to extract insight from Stack Overflow questions and then how to integrate them with developer's tool-chains [161]. In a small study of developers they established that developers found these extra insights helpful when programming. In a similar manner, our work studies developers' struggles through an analysis of the questions they ask on Stack Overflow. Our work adds to the literature by using Stack Overflow not to gain insight into developers, but rather into the usability issues of APIs the developers use.

3.2.2 Usability Issues of Cryptography Libraries

Nadi et al. [109] examined over 1,000 Java cryptography-related questions and developed a set of 5 *obstacles* capturing the developer's problem based on the top 100 questions. Whilst Nadi et al. just looked at Java developers' struggles, our work is broader examining developers' struggles with libraries for multiple languages and systems. They reported a low inter-rater reliability score (kappa = 0.41). We also relate these back to principles to identify underlying issues in the form of usability smells.

Egele et al. looked at the use of cryptographic APIs in Android applications [46]. The study found mistakes in 88% of the apps that used cryptographic APIs. Egele et al. developed a static analysis tool to identify mistakes automatically and proposed three usability guidelines to help developers avoid making mistakes in the future. Mindermann et al. studied the usability of cryptography APIs for the Rust programming language [105]. The study found that, whilst insecure defaults (and defaults in general) do not occur frequently in cryptographic libraries, very few projects warn about depreciated cryptography techniques or encourage developers to use more secure methods. They produced a list of 13 recommendations for cryptography APIs.

Various studies have assessed the usability of specific cryptographic libraries [17, 67, 105] and developed a series of principles that can be followed to improve the usability of cryptographic libraries. For example, Mindermann et al. recommend the following:

> "Use a prominent location to link to the documentation, e.g., at the start page of the repository." [105]

Bloch suggests providing examples so that developers can see how to use the library:

> "Example code should be exemplary. If an API is used widely, its examples will be the archetypes for thousands of programs." [17]

The issues we identify in this paper complement these guidelines by acting as usability smells—usability principles tell us how to make libraries more usable, the smells suggest where users are struggling due to such principles not being observed or implemented.

Studies have also shown that even if a cryptographic library is powerful developers may suggest to use an alternative cryptographic library which, although more usable, may be poorly implemented [1, 67]. For instance, Acar et al. [1] showed that the Python cryptographic library *Keyczar*, despite claiming to be designed for usability, was challenging to use because of poor documentation and lack of documented support for the key generation task. Surveying the developers after their study Acar et al. found that developers frequently found issues with missing documentation and examples in Python cryptographic libraries. We also find in our analysis of Stack Overflow questions that many developers struggle and ask questions due to issues with *missing documentation* and a need for *example code*, alongside several other issues.

If not all cryptography libraries are equally usable, then what issues do developers struggle with when using them? The analysis presented in this paper sheds light on such issues and identifies the usability smells that indicate that usability principles are not being fully observed in the design of a particular library.

3.3 Method

We investigate the following:

1. What issues with cryptographic libraries cause developers to seek help and ask questions on the Stack Overflow question and answer site;
2. How prevalent are the usability issues that we identify in seven cryptography libraries; and
3. What are the usability smells that are indicative of failures to implement Green & Smith's usability principles, and their prevalence in the 3 libraries for which we have sufficient data.

To answer these questions we selected seven cryptographic libraries to examine (Table 3.2), based on their prominence as well as to include a breadth of languages—C, Java, JavaScript and Python—and use-cases. Other libraries exist, but these seven cover a representative sample of what various cryptographic libraries currently look like including those with many users (OpenSSL) to those with only a few (SJCL). Additionally, two of the libraries (NaCl and libsodium) describe themselves as being *usable*, so we would hope to see a range of issues and differences between the usable cryptographic libraries and the not-so usable ones.

We scraped 2,491 questions from Stack Overflow, using the library names as search terms and selecting only questions with a score greater than 1 to help avoid low-value and badly worded questions. Stack Overflow uses a reputation system to help combat spam—users with sufficient reputation[1] are allowed to vote for the usefulness of a question, which becomes the question's *score*.

We conducted a manual review of all 2,491 questions: identifying the underlying issue, capturing common themes between questions, and verifying the validity of the answer. In order to do so, we studied the full description of the question on Stack Overflow to help us pinpoint the core theme of the issue. We also analysed the explanations provided by other developers in the answer section to that question. The questions in our corpus cover a wide time period. Therefore, to address the issue of validity, once we studied the question and any related answers, we reviewed the prominent link of the

[1]At the time of writing: 15 reputation to vote up, 125 to vote down.

cryptographic library to assess whether the question was still valid and unaddressed by the library. For example, if the developer said that they could not find documentation for a specific feature they wished to use, we checked if the documentation was still unavailable in the current version of resources for the cryptographic library. Questions that did not relate to an issue with the library itself, for example, due to users not understanding the behavior of their operating system's dynamic linker (we return to this issue in the Discussion; Section 3.7) or where the developer mistakenly attributed their question to a library, were not considered further. In total, we analysed 2,317 relevant questions.

We used thematic analysis, a qualitative research method used to extract themes from text [21, 34, 138], to identify recurring themes such as the need for documentation, build and compatibility issues within the Stack Overflow questions. We developed our themes by categorising questions, and then reviewing and discussing the categories. We arrived at a set of 16 themes that captured the different issues developer's faced and ascribed a final, single theme to each of the questions we examined. Initially we used multiple themes, however we found only 4 cases where a question had multiple labels, so we simplified and ascribed the theme that best categorised the underlying reason the question was asked. We repeated the categorising process with a regularly selected 10% subset of the questions analysed by a second researcher and calculated Cohen's kappa—a commonly used measure of inter-rater agreement [31]. Cohen's kappa was 0.76 indicating that our coding was consistent between mappers.

3.3.1 Threats to validity

The questions in our corpus cover a period of several years. There is a danger that as time and the cryptographic libraries themselves change that the issues developers face could also change. To mitigate this we validated that usability issue identified in the library were still present in the current version. For example, if we attributed an issue to the documentation being missing, we validated that we still couldn't find the relevant documentation.

There is also the danger that an issue faced by a developer may be due to a particular problem faced solely by that developer and not a more general problem. To mitigate against this we selected questions which had a score greater than one—that is to say that more users of Stack Overflow believed the question to be worthwhile than not. Stack Overflow's reputation system is designed to help remove questions that have already been answered, and those that are of low-value (for example, questions where a developer has not asked a question, or questions where students are attempting to have their coursework answered for them). By selecting only questions with a positive score we help avoid some noise.

During our thematic analysis, each question was mapped to a single theme, with the dominant theme being picked in the case that a question could be attributed to multiple themes. For the most part, however, questions could be ascribed to a single theme and multiple themes were rare so a 1–1 mapping was used for consistency.

To identify the usability smells, we map the issues we identify to the usability principles that library developers should be implementing as identified by Green & Smith [67]. Various others have suggested different principles for developers (as we discuss in Section 3.2). We selected Green & Smith's principles because their principles have not currently been validated, and were themselves a synthesis of other usability research [17] focused on usability and security issues. Other principles could be validated using the same methods and corpus as we have used however, and our dataset is available for comparative studies.

3.4 What usability issues do developers face?

Our thematic analysis reveals 16 usability issues with which developers struggle (Figure 3.3) categorised into 7 themes as shown in Figure 3.1. We discuss each of the issues and give some examples to demonstrate how they manifest in the questions posed by the developers.

3.4.1 Missing information

Missing Documentation

A developer states that they wish to use a function or form a feature that has components supported by the library but cannot find relevant information in the library documentation:

> "So I already know how to specify locations for trusted certificates using `SSL_CTX_load_verify_locations()`.[···] But nothing is mentioned about the trusted system certificates residing in the OPENSSLDIR."

Looking for Example Code

Not all library functionality needs an example, but it can be helpful to document common use-cases. The developer wishes to use a function supported by the library and requests examples of how the function is used. In the question the developer may address the quality of the example code or lack thereof:

> "I'm attempting to run:
>
> ```
> openssl pkcs12 -export -in "path.p12" -out "newfile.pem"
> ```
>
> but I get an error.
>
> ```
> unable to load private key
> ```
>
> How do I extract the certificate in PEM from PKCS#12 store using OpenSSL?"

This differs from *passing the buck* in that the developer has identified the functionality they want to use and made attempt at solving it. They have stated the problem they want to solve and have asked for a example in order to debug their own attempt.

Clarity of documentation

The developer found the documentation or output but found it vague or unclear in describing what exactly it does:

> "How can I interpret openssl speed output?
>
> I ran openssl speed on my Ubuntu computer. [···] what is 'Doing md4 for 3s' mean? does it mean do the whole test for 3 times/seconds? what does '1809773 md4's in 2.99s' mean? what does '8192 size blocks' mean? [···] And the above, last lines of openssl speed md4 output - what does they mean exactly?"

3.4.2 Not knowing what to do

Passing the buck

The developer delegates their question to the Stack Overflow community, even though a quick search for the issue on the library website returns the answer needed:

> "I'm trying to convert the .cer file to .pem through openssl, the command is:

```
openssl x509 -inform der -in certnew.cer -out ymcert.pem
```

> and that's the errors I'm getting:

```
unable to load certificate
```

> What am I doing wrong?"

Rather than find the answer themselves the developer has *passed the buck* and used Stack Overflow to get the answer rather than search existing resources, as reflected in the response:

> "[···] like explained by ssl.com, a .cer file [···]"

Passing the buck differs from other issues, such as *what's gone wrong here*, in that the developer has made no attempt to solve the problem. They have encountered a problem and want someone else to give them the answer rather than work it out or find an existing solution by themselves.

Lack of knowledge

There were many instances where new users struggled with the functions provided by a library due to the lack of knowledge they had about the concepts of cryptography. For example:

> "[···] I'm using OpenSSL to avoid pay for it. I created my certificate this way: [···]
>
> But when I navigate to the website I get an "error" telling me that this is an "Untrusted certificate": The security certificate presented by this website was not issued by a trusted certificate authority."

This lack of knowledge is implicitly highlighted in the answer:

> "What you get from OpenSSL tool is a self signed certificate. Of course it is not trusted by any browser, as who can say you are worth the trust.
>
> Please buy a certificate if you want to set up a public web site [···]"

3.4.3 Not knowing if it can do

Unsupported feature

The developer wants to do something that the library does not support. This may suggest that the library is unclear about what it can and cannot do:

> "Has anybody Implemented ElGamal using OpenSSL or even inside?"

Borrowed mental models

The developer is trying to take a mental model about how one library works and apply it to different one:

> "How to recreate the following signing cmd-line OpenSSL call using M2Crypto in Python?:
>
> This works perfectly in command-line, I would like to do the same using M2Crypto in Python code.
>
> [···]"

The developer has tried to apply concepts from one library to another and has become confused when that doesn't work. This differs from *passing-the-buck* in that they are not unwilling to learn, they just don't know that the concepts differ. Passing-the-Buck is where a developer doesn't know how to use a library and tries to get someone else to tell them. They don't care about learning and just want to be told what to do.

3.4.4 Programming is hard

Abstraction issue

The developer needs help with an abstraction provided by the library. They've seen the documentation but they lack knowledge of the underlying abstraction to understand it. They need more help:

> I am trying to get my head around public key encryption using the openssl implementation of rsa in C++. Can you help? So far these are my thoughts (please do correct if necessary) [···] I see these two functions: [···] If Alice is to generate *rsa, how does this yield the rsa key pair? Is there something like rsa_public and rsa_private which are derived from rsa? Does *rsa contain both public and private key and the above function automatically strips out the necessary key depending on whether it requires the public or private part? [···]

What’s gone wrong here?

The developer has tried to use the library but has failed. They have given a specific example and asked Stack Overflow to suggest what has gone wrong:

> “Here is a certificate in x509 format that stores the public key and the modulo:
>
> `const unsigned char *certificateDataBytes = /*data*/;`
>
> Using OpenSSL and C, how can I convert it into an RSA object? I’ve tried several methods but I can’t get it to work in `RSA_public_encrypt`”

API misuse

API Misuse represents questions where the developer incorrectly uses a function and they are corrected by another developer, usually supported with an explanation of the answer. For example:

> “[···] I’m trying to build a handshake protocol for my own project and am having issues with the server converting the clients RSA’s public key to a Bignum. It works in my client code, but the server segfaults when attempting to convert the hex value of the clients public RSA to a bignum.”

In the response to the question, the correct use of the function is explained:

> “RSA new() only creates the RSA struct, it does not create any of the bignum objects inside that struct, like the n and e fields. [···]”

3.4.5 Usage Issues

Should I use this?

Developers have tasks and features in mind for which they want to know whether they should use a specific library function or not—or if there are two or more functions, which one should they use? In other cases, developers want to know whether the choices they make regarding the security of their application are appropriate:

> “I’m trying to build two functions using PyCrypto that accept two parameters: the message and the key, and then encrypt/decrypt the message.
>
> I found several links on the web to help me out, but each one of them has flaws:
>
> [···] Also, there are several modes, which one is recommended? I don’t know what to use :/”

This differs from *missing documentation* where the developer is searching for specific API documentation.Here, they are unsure about which part of the API they want to use in the first place.

How should I use this?

In contrast with *Should I use this*, in such cases the developer knows what they want to use, but is confused about some of the parameters involved:

> "How to compute RSA-SHA1(sha1WithRSAEncryption) value with OpenSSL?"

3.4.6 System issues

Build issues

To use an API developers must first build it (and run tests). This causes problems for the developer:

> "Error compiling OpenSSL with MinGW/MSYS"

> "How to build OpenSSL to generate libcrypto.a with Android NDK and Windows"

Performance issues

The developer wants to use a library but finds that it's performance may not be enough for their use-case. They seek help in optimizing their use of the library:

> "[$\cdots$] profiling has revealed [$\cdots$] 40% of my library runtime is devoted to creating and taking down HMAC_CTX's behind the scenes. [$\cdots$] How do I get rid of the 40% overhead on each invocation in a (1) thread-safe / (2) resume-able state way? [$\cdots$]"

3.4.7 Issues across space and time

Compatibility issues

The developer is struggling to integrate the library in question with another platform or library. For instance, out of the 2,022 questions pertaining to OpenSSL, 244 were related to compatibility issues:

> "Encrypt in C# using OpenSSL compatible format, decrypt in Poco:
>
> I'm trying to encrypt (aes-128-cbc) in Win OS using a OpenSSL compatible format and decrypt on Linux OS using Poco::Crypto that is a wrapper of OpenSSL."

Deprecated feature

The developer is trying to do something the library once supported, but doesn't know that the latest version has deprecated it:

> "After a few days of scouring the internet and openssl docs i've hit a wall [$\cdots$]."

Table 3.3: The 16 issues identified through a thematic Analysis of Stack Overflow Questions.

Usability Issue	Description
Missing Documentation	The cryptographic library does not have documentation available to address the issue.
Example Code	The developer asks for code examples to learn how to use a specific feature of the library or to learn how to implement some behavior. An example is either missing or lacking somehow.
Clarity of Documentation	The library has documentation for the developer's issue, but it is unclear or lacking additional information. The developer asks for clarification.
Passing the buck	The developer asks Stack Overflow, even though documentation regarding their issue has been given. Also questions where they ask a simple question which they answer themselves.
Lack of Knowledge	The developer does not have foundation level cryptography knowledge. The developer is new to cryptography as a subject and, in turn, the features of the cryptographic library.
Unsupported Feature	The cryptographic library does not support a security feature the developer wants to implement.
Borrowed Mental Models	The developer requests a mapping of a functionality between cryptographic libraries.
Abstraction Issue	Issues addressing the level of abstraction provided in the code of the cryptographic library. The developer wants a more detailed explanation than is provided by the documentation.
What's gone wrong here?	The developer has code that looks like it should work, but fails—they are looking for an explanation why.
API Misuse	The developer has incorrectly used a specific feature from the cryptographic library.
Should I use this?	The developer says what they wish to implement and asks which methods would be most apt to use.
How should I use this?	The developer does not understand how to correctly use a feature or its various parameters.
Build Issues	Issues related to the setup of the cryptographic library and running provided tests.
Performance Issues	Issues regarding the performance of the cryptographic library.
Compatibility Issues	Issues related to integrating features from the cryptographic library with other libraries and tools.
Deprecated Feature	Issues addressing that a specific feature is not working, later to conclude that the feature is deprecated.

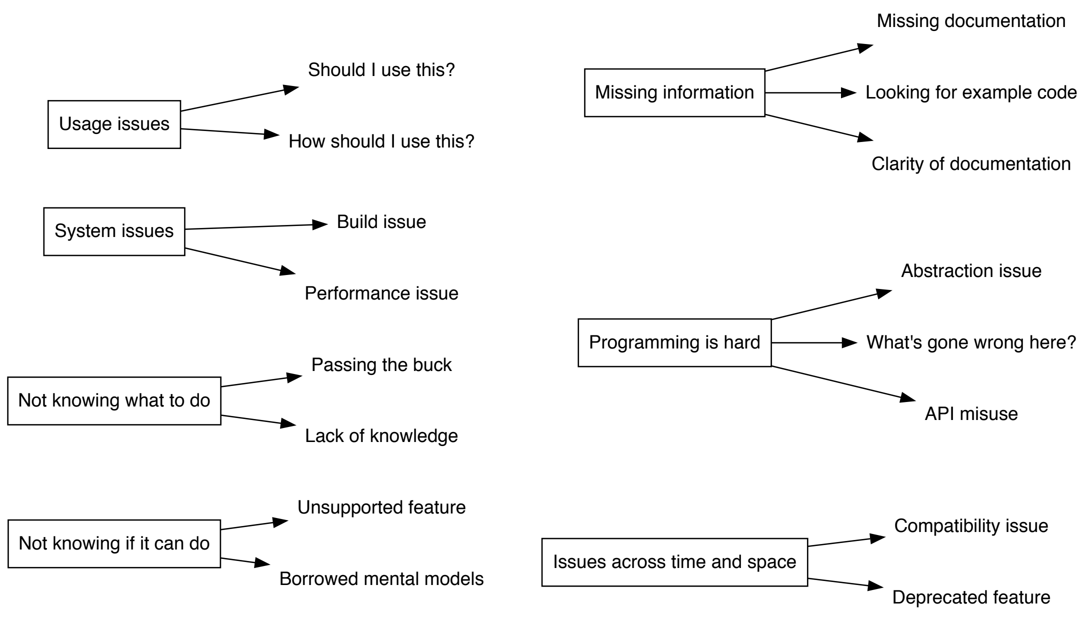


Figure 3.1: Categorisation of the 16 issues identified through the thematic analysis.

3.5 How widespread are the issues across the seven libraries?

Table 3.4 shows the number of times each issue appeared during our thematic analysis for each cryptographic library; and suggests common issues across the libraries. *Missing Documentation* is a common issue: it suggests that developers face an issue in the first stages of using a cryptographic library as they are unable to locate documentation to support them. For instance, SJCL provides the code of each of its functions as its only developer support resource, and so can be made much stronger if they considered adding documentation to support the functions provided.

Passing the Buck and *Lack of Knowledge* highlight issues associated with developer behaviors instead of the cryptographic libraries themselves. Passing the Buck issues are common showing that developers have a tendency to pose questions on Stack Overflow, while the resources addressing the very questions are provided by the library and easy to locate. Many instances are recorded under the OpenSSL library, along with Bouncy Castle and Crypto-JS. Bouncy Castle and PyCrypto have a high percentage of questions associated with Lack of Knowledge. Developers address their lack of knowledge in their questions and request support in learning cryptographic concepts in order to use functions from these libraries.

There are many occasions where the developer has a specific feature in mind for a project and wants to know how to securely implement this feature into the project. The reason the number of questions defined under *How should I use this?* is high for OpenSSL, for example, may be because the developer believes that other developers have already implemented the feature they had in mind. So the developer resorts to finding the specific implementation on developer community sites such as Stack Overflow instead of building their feature using the documentation provided by the cryptographic library. This could also explain why there are many questions where developers show an example of their broken code and request guidance with debugging. The answers usually come in the form of task-based examples of how to correctly implement, something to which the developers respond well.

Issue	OpenSSL	Libsodium	NaCl	Bouncy Castle	SJCL	Crypto-JS	PyCrypto
Missing Documentation	256 (13%)	3 (9%)	5 (12%)	31 (17%)	4 (27%)	2 (6%)	7 (4%)
Example Code	128 (6%)		1 (2%)	10 (5%)	2 (13%)		4 (3%)
Clarity of Documentation	92 (5%)		3 (7%)	2 (1%)			6 (4%)
Passing the buck	136 (7%)	2 (6%)	4 (10%)	22 (12%)	4 (27%)	17 (49%)	10 (6%)
Lack of Knowledge	44 (2%)	6 (19%)	3 (7%)	19 (10%)		4 (11%)	17 (11%)
Unsupported Feature	24 (1%)		1 (2%)	5 (3%)			7 (4%)
Borrowed Mental Models	56 (3%)		2 (5%)	1 (1%)			
Abstraction Issue	40 (2%)		2 (5%)	2 (1%)	2 (13%)	2 (6%)	10 (6%)
What's gone wrong here?	259 (13%)	1 (3%)	2 (5%)	24 (13%)		3 (9?%)	16 (10%)
API Misuse	11 (1%)		1 (2%)	6 (3%)			7 (4%)
Should I use this?	84 (4%)		8 (19%)	19 (10%)		1 (3%)	8 (5%)
How should I use this?	80 (4%)			10 (5%)		2 (6%)	4 (3%)
Build Issue	362 (18%)	7 (22%)	3 (7%)	15 (8%)		3 (9%)	57 (36%)
Performance Issue	20 (1%)		1 (2%)				
Compatibility Issue	244 (12%)	7 (22%)	6 (14%)	8 (4%)	3 (20%)	1 (3%)	5 (3%)
Deprecated Feature	20 (1%)			9 (5%)			1 (1%)
Not Relevant	166 (8%)	6 (19%)		2 (1%)			

Table 3.4: Count of the number of Stack Overflow Questions attributed to each usability issue per library. Zero counts ommited.

Other than Missing Documentation, developers also highlight difficulties they have while setting up OpenSSL and running the provided tests. Reviewing the questions, we see that developers have projects in mind and intend to implement OpenSSL with other platforms they are using. This raises many questions associated with *Compatibility*. Developers find it very difficult to integrate OpenSSL with other platforms, a particularly pertinent issue as OpenSSL is widely used—and for large-scale projects. However, having compatibility issues makes OpenSSL less usable as developers cannot easily reconcile implementation of security requirements with other requirements for their projects.

3.6 Usability Smells

Having identified the above issues, we map them on to Green & Smith's 10 principles (shown in Figure 3.1) in order to identify the usability smells that are indicative of one or more of the principles not being fully observed. The purpose of these smells is not to identify usability issues with a library

Whiff	Issue	Abstract	Powerful	Comprehensible	Ergonomic	Intuitive	Failing	Safe	Testable	Readable	Explained
Need a super-sleuth	Missing Documentation				●						
	Example code				●		●				
	Clarity of documentation				●		●				
Confusion reigns	Should I use this?			●		●					
	How should I use this?	●	●			●					
	Abstraction issues	●				●					
	Borrowed mental models				●						
Needs a post-mortem	What's gone wrong here?				●	●				●	●
	Unsupported feature					●					
	API misuse						●	●			
	Deprecated feature						●				
Doesn't play well with others	Build issues								●		
	Compatibility issues		●								
	Performance issues										

Table 3.5: Mapping between developer issues and Green & Smith principles.

early, but rather to guide work to improve a library's usability based on where developers appear to struggle between releases of a library as part of the software lifecycle. We note that Green & Smith's principles are written in a positive manner—for example:

> "Integrate cryptographic functionality into standard APIs so regular developers do not have to interact with cryptographic APIs in the first place."

In contrast, the issues we identify from the thematic study are written in a negative context—for example:

> "Missing Documentation: The cryptographic library does not have documentation available to address the issue."

The two viewpoints however are linked—if a library developer fails to fully implement a usability principle, then we might expect to see questions indicating that the library users are struggling with one of the usability issues we identify. Our mapping between usability principles and usability issues is presented in Table 3.5.

For each issue we identified, we considered whether it would indicate failing to implement one of Green & Smith's principles. We did not map the *lack of knowledge* or *passing the buck* issues as these are attributable to specific developer behaviors and do not represent failures to implement usability within a library, and so do not map to Green & Smith's principles. For example the *borrowed mental* *model* issue is present when a developer expects one library to work similarly to another; this is mapped to the *ergonomic* principle as it indicates a failure to not break the developer's paradigm.

Based on this mapping, we identify four usability smells. In the same fashion as Fowler [55], we describe them as *whiffs*.

3.6.1 Needs a super sleuth

Issues at play: Missing documentation, Example code, Clarity of documentation

You whiff this when documentation is not provided, unclear or there is a lack of example code available to how to use the library. The information needed to achieve the task you plan to to undertake is hard to find or understand in a way that can help you use the library easily. You need to be a super-sleuth to find the documentation and decipher its meaning.

By not breaking the developer's paradigm (the ergonomic principle), developers can intuitively make use of the library for tasks with fewer references to the documentation or example code. By improving the visibility of errors and showing them early (the failing principle) developers can quickly understand when something is wrong, how to solve the issue, and fix it themselves.

3.6.2 Confusion reigns

Issues at play: Should I use this?, How should I use this?, Abstraction issue, Borrowed mental models

You can catch a whiff of this when developers are designing and prototyping their programs—during this process they are trying to decide whether this is the right library to use and how to start using it. They are unclear about how to use the library, perhaps having confused some concepts or borrowed a mental model they have for another library that isn't relevant here.

By making the library easy to use even without documentation (intuitive principle) a developer can quickly work out if they should use a library for their program, and how to use it and understand the abstraction it provides. If it is easy to learn (comprehensible principle) they can quickly evaluate it. If it uses standard APIs (abstraction principle) they can quickly learn how to use without engaging with the cryptographic primitives under the functions. By not breaking the developer's paradigm (ergonomic principle) they can reuse existing mental models about how similar libraries behave.

3.6.3 Needs a post-mortem

Issues at play: What's gone wrong here?, Unsupported feature, API misuse, Deprecated feature

If the *confusion reigns* whiff concerns the smells pre-coding, then this whiff occurs after they have written some code. The developer has used the library only to realise that something has gone wrong. Either they have used the library incorrectly or they are struggling to work out if there is an issue with the library itself. Perhaps an update to the library has broken their code, a functions has been deprecated for example, or the documentation has led them to believe that it can do something it cannot—either way the code needs a post-mortem.

By not breaking the developers model (ergonomic principle) developers can quickly guess whether their code has errors and work out what's gone wrong. If the library is easy to use (intuitive principle) then this aids with debugging what's gone wrong as well as figuring out the capabilities of the library. Being hard to misuse (failing principle) avoids API misuse, preventing API designers from deprecating APIs without a warning, and preventing developers from using deprecated APIs as well. If the defaults are safe and sensible (safe principle), then developers may avoid the complex API features and their potential misuses. Making the code easy to read (readable principle) means that if a developer needs to identify the origins of a bug by diving into the code, they can do so with the minimum of fuss. Finally by helping developers with end-user interaction (explained principle), library designers can ensure developers use the library in a standard way hence avoiding the need to ask if other people have done things the same way or differently.

3.6.4 Doesn't play well with others

Issues at play: Build issue, Compatibility issue, Performance issue

If a library is going to be easy to use, developers should be able to use it in the first place. This smell occurs when the library will not build nor integrate with other libraries and build systems, and is a resource hog without providing a clear explanation why.

This smell doesn't appear to be particularly well covered by Green & Smith's principles. By adding a testing mode with only a subset of the features active (testable principle) developers can avoid having to build all the dependencies and get the library built for early testing and prototyping. By making the library powerful enough to satisfy security and non-security requirements (powerful principle) developers can more easily integrate it with other less flexible libraries.

> "Firstly, most developers using a security API do not have a firm grasp on the cryptographic or security background and thus would be hard pressed to explain to the end-user what went wrong" [67]

Perhaps by extending this principle to include not just *why things go wrong*, but also *why things take so long* this additional issue could be covered by Green & Smith's ten principles. Alternatively the *powerful* principle could be extended to cover not just the developer's primary security and non-security functionality requirements, but also cover the performance aspects.

3.7 Discussion

With four whiffs established, Table 3.6 describes how prevalent these issues are in 3 of the cryptographic libraries—OpenSSL, Bouncy Castle and PyCrypto. For the remaining 4 libraries we lack a sufficient

Whiff	Issue	OpenSSL	Bouncy Castle	PyCrypto
Needs a super sleuth	*Whiffiness factor*	10% ●	11% ●	4% ●
	Missing documentation	13%	17%	4%
	Example code	6%	5%	3%
	Clarity of documentation	5%	1%	4%
Confusion reigns	*Whiffiness factor*	3% ●	6% ●	4% ●
	Should I use this?	4%	10%	5%
	How should I use this?	4%	5%	3%
	Abstraction Issue	2%	10%	6%
	Borrowed mental models	3%	1%	0%
Needs a post mortem	*Whiffiness factor*	10% ●	11% ●	8% ●
	What's gone wrong here?	12%	13%	10%
	Unsupported feature	1%	3%	4%
	API misuse	1%	3%	4%
	Deprecated feature	1%	5%	1%
Doesn't play well with others	*Whiffiness factor*	11% ●	5% ●	22% ●
	Build issue	18%	8%	36%
	Compatibility issue	12%	4%	3%
	Performance issue	1%	0%	0%

Table 3.6: What whiffs can you smell on each library? Percentages of the questions for each library that were mapped to each issue are shown, along side a *Whiffiness factor*, based on the weighted average, that indicates how strong the smell is:

●: particularly pungent (weighted average >10%);

●: merest whiff (weighted average $\geq 2, \leq 10\%$).

volume of questions to make any meaningful statement about the issues with which the library's users may struggle. However, for these 3 libraries we have 2,022, 185 and 160 questions respectively and so can consider where the core challenges for developers using these libraries may lie. We include the libraries with fewer questions in our thematic analysis in order to reduce skew towards the issues prevalent in the more frequently queried libraries, however we lack the volume of questions required to suggest what the challenges for developers are in the 4 remaining libraries. We do not claim that the libraries are at fault—rather we suggest what the most frequent issues that some developers struggle with when using them are—and where the greatest improvements in usability may be made. Future work should explore and find the underlying cause for the usability smells and establish *why* developers appear to be struggling.

For each library we add a *Whiffiness factor* (based on the weighted average of the percentage frequency of the issues associated with each whiff). All the libraries we looked at show signs of *needing a super sleuth* with OpenSSL and Bouncy Castle users especially struggled with missing documentation.

Despite this the overall whiffiness of this smell appeared to be low, as there were fewer questions over all 3 libraries associated with these issues—this suggests that documentation may be improving; and whilst documenting more of the library and giving more examples will help users, there may be greater improvements in usability to be made elsewhere.

As for the *confusion reigns* whiff, again, the libraries all seem to show some signs of it—with the issue being particularly pronounced for Bouncy Castle, where we saw many developers asking whether it was appropriate to use this library, and having particular issues with the abstractions it provides. This is somewhat surprising as, at least for the Java version, Bouncy Castle integrates with the Java Cryptography Architecture which provides a standard API for libraries providing cryptography functionality. Bouncy Castle also provides its own API, and supports languages other than Java—perhaps offering too much choice confuses developer as to the parameters for a specific version. Focusing on the intuitive and comprehensible principles, i.e., by making the library easier to learn and understand without the need for expertise, should help reduce this smell.

The *doesn't play well with others* usability smell was present for all three libraries. OpenSSL and PyCrypto in particular struggled with *build issues*, whereas Bouncy Castle (which is available as a precompiled JAR file) had fewer issues associated with building the library. Integrating software into systems is known to be difficult [61], but offering pre-built images seems to go some way to mitigating this. Building the library is just the first step for OpenSSL however, as the library has to be linked into the final compiled program. When mapping the Stack Overflow questions, we saw several examples of developers asking about the dynamic linker. For instance:

> "I am using OpenSSL in my project.library is detecting but getting some errors like below:
>
> `Error:(23) undefined reference to 'RSA_generate_key'` [···]
>
> I included appropriate .so files in appropriate folder. I am not getting reason behind the undefined reference error.please help me to solve this issue."

These are not library usability issues as they represent a misunderstanding about the host-system and tools rather than the library itself. So we did not map them to an issue (the question in the specific example above was resolved by the developer updating their Makefile). The issue was common enough, however, that we believe that there may be a serious usability issue integrating libraries with systems, and a gap in the literature in looking into these issues. Further work is needed to map out what these issues are, how common they are and what we can do to mitigate them.

The final usability smell we identified, the *needs a post mortem* smell, was prevalent in the OpenSSL, Bouncy Castle and PyCrypto libraries. For these libraries the biggest contributing issue to this usability smell was that of developers trying to establish what had gone wrong with their programs. Making the code more readable and the libraries more intuitive to use even without documentation should help to make the debugging process easier and mitigate this smell. OpenSSL and Bouncy Castle are lower-level than other cryptography libraries providing greater access to their internals and cryptographic primitives. For these libraries we would expect an increase in the number of questions by developers trying to debug the code, simply because they wrap things up less into high-level APIs and offer more scope for developers to make a mistake. Perhaps then it is not unreasonable to expect lower-level libraries to display this issue more than the higher-level ones.

OpenSSL in particular, has been criticised in the past for being hard to use [1, 85, 162]. Kamp in particular argued for someone to:

> "Please Put OpenSSL Out of Its Misery. OpenSSL must die, for it will never get any better." [85]

Our analysis associates strong usability smells with OpenSSL—in particular it seems that developers struggle a lot debugging it and in finding the documentation. Ignoring those issues, however, it is similar to the other cryptographic libraries and sometimes a little bit better (it seems better at abstraction, describing its parameters, for example)—at the very least both Bouncy Castle and PyCrypto appear harder to debug.

3.8 Conclusion

Through our analysis of a substantial corpus (2491) of questions from Stack Overflow, we found 16 issues and four usability smells that suggest when developers are struggling. By linking these usability smells to the 10 usability principles defined by Green & Smith [67], we can suggest how to improve cryptographic libraries and make them more usable for developers. Our study offers evidence to validate parts of Green & Smith's heuristics, but also highlights issues that were missed. Their usability principles suggest ways to mitigate most of the issues we identify; however issues associated with the *doesn't play well with others* smell (in particular *build* and *performance* issues) suggest the need for an additional principle to help cover these issues.

Our usability smells capture the general challenges developers face when using cryptographic libraries. Not all libraries smell the same, and improvements to *usable* cryptographic libraries appear to be beneficial with fewer usability smells. By identifying these usability smells we can find the challenges faced by developers and help improve usability. Libraries will perhaps always show signs of these usability smells given the challenges of catering for the requirements of a wide and diverse set of developers and applications; but by integrating usability principles we can at least make them less so.

In this chapter we identified what challenges developers face as they try to use different cryptographic libraries. The next question we ask is *why* do the developers face the challenges that they do? Cryptographic libraries can differ from each other in terms of design, and the levels of abstraction used in their design but the common factor between them is the cryptographic concepts upon which their libraries were based. We can say with great confidence that the cryptography experts behind the design of these cryptographic APIs understand how a concept like Public Key Cryptography works but can this be said about the developers who use these APIs? Even high-abstraction level cryptographic APIs like Libsodium frequently use terminology like *encryption*, *digital signatures*, and *hashing* in their documentation but do developers know what these terms mean, if so do they have any misconceptions that prevent them from correctly using the cryptographic API? We study the relationship between developers and the concept of Public Key Cryptography by eliciting their mental models as they are tasked with designing a system that requires them to achieve the conditions achieved by Public Key Cryptography. We compare their mental models against how the cryptographic concept actually works to pinpoint the exact points at which misconceptions occur and where their mental models begin to deviate from the correct way in which Public Key Cryptography works.

Chapter 4

Developer Mental Models of Public Key Cryptography

Developers use cryptographic libraries to secure their applications. The APIs of these cryptographic libraries are difficult to use. Even with abstraction built into the API design of newer cryptographic libraries to improve usability, developers are still required to have some fundamental knowledge of cryptographic concepts before using them. Through an interview format based on the think-aloud protocol, we ask 20 participants to design a system that achieved the conditions met by Public Key cryptography (PKC). During the interview, participants also perform a diagramming exercise to better explain how their system achieves the conditions of confidentiality, authentication, and integrity. We elicit the developers' mental model through our analysis using the straussian grounded theory. As this is a protocol design exercise, developers also show their problem-solving abilities which are also captured and analysed. We hypothesise that there are misalignments between the developers' mental model and how Public Key Cryptography actually works, and that these misalignments are a primary cause for the usability issues developers face when trying to use cryptographic APIs.

4.1 PKC Protocol Design With Software Developers - A Mental Model Approach

4.1.1 The Original Mental Models of Public Key Cryptography

The term *mental model* was first coined by Craik in 1943 in his book titled *The nature of explanation*.

> "A mental model is a physical working model which works in the same way as the process it parallels." [32]

Many mental models were used during the initial discoveries made in attempts to address the challenges of Public Key Cryptography.

During the 60's there was a need for a secure method of communication to allow for the exchange of sensitive information for military purposes [49]. This method needed to address two challenges [44]:

- The first challenge was that of key distribution: For two people who have never met before and want to communicate with each other *privately* through conventional cryptographic methods, they must somehow agree on a key that will be known to no one but themselves [44].
- The second challenge was to produce a *digital* signature: Could a method be devised that would present the recipient of a digital electronic message with a way of showing to others that the message had come from a particular person, just as a written signature at the end of a letter allows the recipient to hold the author to their word [44]?

Whitfield Diffie, of Diffie-Hellman, realised that if Bob wanted to be sure that a message came from Alice, Bob could challenge Alice's identity by posing questions that, although Bob would be unable to answer, could be judged for correctness. Through a generalisation of the *trap-door one-way function*, someone in possession of a secret piece of information could compute the function's inverse. This process later became a way to authenticate someone's identity, like a signature, a *digital signature*. The idea of inverse functions led to the discovery of Diffie's public-key cryptosystem, where keys come in pairs and perform the inverse of one another. These keypairs had two properties [44]:

1. Information encrypted with one key can be decrypted by another [44].
2. Given the public key, it is infeasible to discover the secret key [44].

To explain the separate functions of the keys in the keypair, Diffie used the mental model of a *telephone directory*. Diffie explained how users of the communication system, like Alice and Bob, could list their public keys in a *telephone directory* along with other details such as their names and addresses. Diffie continues on to show how to use this telephone directory [44]:

1. "One subscriber can send a private message to another by simply looking up the addressee's public key and using it to encrypt the message. Only the holder of the corresponding secret key can read such a message; even the sender, should they lose the plaintext, is incapable of extracting it from the ciphertext [44]".
2. "A subscriber can sign a message by encrypting it with their own secret key. Anyone with access to the public key can verify that it must have been encrypted with the corresponding secret key, but this is of no help to the recipient in creating (forging) a message with this property [44]".

Merkle, at the time a student to Lance Hoffman's computer security course at U.C Berkley, addressed the challenge of public-key distribution, described by Merkle as 'secure communication over insecure channels' [101]. Merkle proposed a simple solution: to communicate the cryptographic key between two people by hiding it in a large collection of puzzles. Merkle explains his puzzle based mental model in more detail [44, 101]:

1. “Alice manufactures a million or more puzzles and sends them over the exposed communication channel to Bob. Each puzzle contains a cryptographic key in a recognisable standard format. The puzzle itself is a cryptogram produced by a block cipher with a fairly small key space. As with the number of puzzles, a million is a plausible number [44, 101]”.
2. “When Bob receives the puzzles, he picks one and solves it, by the simple expedient of trying each of the block cipher’s million keys in turn until he finds one that results in plaintext of the correct form. This requires a large but hardly impossible amount of work [44, 101]”.
3. “In order to inform Alice which puzzle he has solved, Bob uses the key it contains to encrypt a fixed test message, which he transmits to Alice. [44, 101]”.
4. “Alice now tries her million keys on the test message until she finds the one that works. This is the key from the puzzle Bob has chosen [44, 101]”.

A report by Bell Telephone Laboratories from 1944 presented an idea for secure telephone speech [93]. The report proposed that noise should be added to the sender’s speech by the recipient. The recipient could remove the noise once they received the message, since they knew what it was. The noise was created by using the negative of the speech signal itself, this meant that no signal could be detected as the message and it’s negative essentially canceled out, but the received signal was left unimpaired. Although this proposal had some practical disadvantages that prevented it from being used, it inspired one James Ellis, a mathematician at GCHQ, who realised that secure communication was theoretically possible if the recipient took part in the encipherment. This breakthrough lead to the theory behind the *non-secret encryption* protocol in 1973, now known as Public Key Cryptography [47–49].

Ellis addressed the encipherment process initially in 1970 with the *existence theory*. The idea being that the encipherment process could be represented through a look-up table, where the cipher text is the contents of the table. Although this table would be incredibly large, in principle, it could be constructed. Based on this idea, Ellis developed a proof that satisfied the initial challenges [44] and demonstrated that non-secret encryption was possible. Although this proof did not come to fruition due to flaws in Ellis’ number theory, it was later realised for practical purposes by Clifford Cocks, which was followed by the RSA algorithm that we know today.

There is a mental model commonly associated with Ellis’ discovery of Public Key Cryptography. The nature of inverse operations seen through the non-secret encryption protocol is represented by a key and padlock, and how the act of locking and unlocking are inverse functions.

1. Alice can buy a key and padlock pair. Alice keeps the key and sends the padlock open to Bob.
2. Bob can write a message and close the padlock, locking the message. This locked message is sent back to Alice.
3. Alice can access the message by opening the padlock by unlocking it with her key.

4. Alice's key is only accessible by her, where as open padlocks are available to anyone who wants to talk to Alice. No keys are exchanged.

4.1.2 The Need to Understand Developers' Mental Models of PKC

Public Key Cryptography is integral to the security engineering of the applications written today and is a functionality commonly offered by cryptographic libraries. As shown in Chapter 3, the cryptographic APIs offered by these libraries are difficult to use. With the aim of becoming more usable, new cryptographic APIs are designed with a high-level abstraction with the purpose of reducing unintentional cryptographic misuse by developers. The abstractions represented through the cryptographic API are based on the API designers' mental model, their understanding of how cryptographic processes work. Although different cryptographic APIs vary from each other and represent processes differently, they stem from the same cryptographic concepts such as Public Key Cryptography, and so offer the same functionality. This cannot be said about when comparing a cryptographic process and a developer's understanding of that process. The developer's level of cryptographic expertise can consist of misconceptions, unaddressed but add to a false sense of confidence before using a cryptographic API. Acar et al. found that 20% of the solutions written by developers, for common cryptographic tasks, were believed to be functionally secure when in reality they were not [1]. These unaddressed misconceptions based on the developers' mental models can clash with the way cryptographic processes actually work. This can lead to many challenges when trying to use a cryptographic API because although the API designers' mental model, used to influence the abstractions in the cryptographic API design, may deviate from the original key & padlock mental model [47], it is still derived from the key & padlock model. Whereas the developers' mental model of cryptographic concepts may be non-existent if they have never performed security tasks before.

The developers' mental models are challenged from the moment they go onto the cryptographic libraries' website. The developer is introduced to a variety of terminology such as *encryption/decryption*, *confidentiality*, *keypairs*, *authentication*, *tampering*, and *hashing* [52]. Existing research has looked into mental models held by end-users when asked about encryption [168]. Wu et al. elicited 4 mental models of encryption through a series of 19 semi-structured phone interviews. The mental models fell under models of access control and models resembling symmetric encryption [168]. The study also showed that participants believed in the idea that people who use encryption for personal reasons were involved in illicit or immoral activities, or possible paranoid. Instead of asking a study group of developers what they think Public Key Cryptography is and how does it work, we present the developer with the challenges for which Public Key Cryptography was proposed and ask them to design a system that satisfies these conditions.

Through our study we elicit, analyse, and present mental models held by developers with regards to Public Key Cryptography. We also study the developer's thought process during the protocol design exercise to identify factors that influence their mental models. Finally, we present a comparison of the developer's protocol design against the way PKC is actually performed. We also compare and contrast the developer's mental models to mental models seen during the initial discoveries at the time PKC was first developed.

4.1.3 Why did we choose Public Key Cryptography?

Public Key Cryptography is a core component of many applications and security networks in use today. Furthermore, Public Key Cryptography consists of many individual cryptographic concepts such as encryption & decryption, hashing, and the use of keypairs. The development of Public Key Cryptography and the mental models used at the time are very well documented, which allows us to compare the mental models we elicit as part of our study with the mental models used back then. This is not only true for mental models but also for the developer's thought process captured by our methodology, where we identify the thought process and the considerations that developers have before making decisions about their design. The process of Public Key Cryptography can be broken down to address the conditions of *confidentiality*, *authentication*, and *integrity*, concepts that serve as a form of abstraction themselves, allowing us to present the analysis more clearly as we can analyse the developers' perceptions by these three concepts. Public Key Cryptography is a protocol in itself, proving to be more of a task for developers than simply asking them what they think Public Key Cryptography means. The steps of Public Key Cryptography serve as a way to study the developers' proposed design and pinpoint exactly where they deviate from the correct path, if they deviate at all. Public Key Cryptography satisfies the following three conditions:

1. *Confidentiality*: If Alice wants to send a message to Bob, Alice wants to be sure that only Bob can read the message. The message cannot be read by Eve.

2. *Authentication*: If Bob receives a message he expected from Alice, Bob needs to know that Alice did indeed write the message and that it was not written by Mallory pretending to be Alice.

3. *Integrity*: After receiving a message from Alice, Bob needs to be sure that the message has not been tampered with in transmission.

The following pseudo notation explains how PKC achieves these three conditions:

```
Confidentiality:
Alice: APrK, APuK
Bob: BPrK, BPubK

Alice – {Message}BPuK -> Bob
Bob – {{Message}BPuK}BPrK = Message
Authentication:
Alice: APrK, APuK
Bob: BPrK, BPubK

Alice – {{Message}APrK}BPuK -> Bob
Bob – {{{Message}APrK}BPuK}BPrK
= {Message}APrK
{{Message}APrK}APuK
= Message
Integrity:
Alice: APrK, APuK
Bob: BPrK, BPubK

Alice – {Message,{{Message}H}APrK}BPuK -> Bob
Bob – {{Message,{{Message}H}APrK}BPuK}BPrK
= Message,{{Message}H}APrK
{{{Message}H}APrK}APuK
= {Message}H
Message
{Message}H

If({Message}H = {Message}H){
Integrity exists.
Else{
Message has been tampered with.
}
```
[a]

[a] Alice's Private & Public keys represented using APrK & APuK. Bob's Private & Public keys represented using BPrK & BPuK. Hash function represented using H.

4.1.4 Research Questions

We elicit, analyse, and present developers' mental models to help answer the following research questions:

RQ1: What thought processes does the developer exhibit while designing the security protocol to help solve for the conditions of PKC?

When tasked with the challenge of designing a system through which Alice can send a message to Bob securely, we question how the developer's ability to consider threats, strategise and evaluate solutions against the identified threats impact their design process and ultimately their mental models.

RQ2: What are the developers' mental models?

While performing the task of designing a secure system that addresses the conditions of PKC, we elicit the developers' mental models that they utilise to help solve for the conditions of *confidentiality*, *authentication*, and *integrity*.

RQ3: How do the developers utilise their mental models in the protocol design process?

Later in the chapter, we present our grounded theory and explain how the developer utilises their mental models and how their mental models are impacted by the other variable components of the design process.

RQ4: What are the misconceptions seen in the developers' mental models and how do they deviate from the standard path of PKC?

The analysis of the developers' mental models shows a series of abstractions and misconceptions associated with the concept of Public Key Cryptography.

4.2 Method

4.2.1 Recruitment & Ethics

Our baseline for recruiting participants were those who had a fundamental understanding of cryptography and could read and write in C. Of the 20 participants, in our study group, 9 participants were recruited through LinkedIn, 8 participants were recruited through universities, and 3 participants were recruited through industry. The 8 participants recruited through universities were introduced to cryptography modules as part of their degree along with modules in C programming, while the remaining participants were software developers who performed security tasks as part of their jobs.

The advertisement for our study was sent to the lecturers of these modules at the universities on the shortlist, with the request that they share the advertisement with their students. The lecturers shared the advertisements through group emails and student boards. On LinkedIn, we joined groups related to cryptography and C programming and asked the administrator for their permission before sharing our advertisement with the rest of the group. We also reached out to industrial connections and shared our advertisement with them as well.

Once someone showed their interest in participating in the study, we gave them a participant information sheet and a consent form. The participant information sheet gave an outline of the study and explained that no personal information would be collected. The interview would take place using Zoom.us and we used browserboard.com, an interactive online whiteboard, for the diagramming exercise. After signing the consent form, we arranged a date and time for the interview and we also asked the participant to fill an initial questionnaire.

Ethical approval for the study was requested from and granted by the University of Bristol, Faculty of Engineering Ethics Committee. No personal data was collected, and demographic data was deleted after coding and validation. Data is available by request.

4.2.2 Study Design

At the beginning of the interview, we introduced the study and clarified any questions the participant had about the task. We then sent them a link to an interactive whiteboard, through browserboard.com. On the whiteboard, the participant was shown the three conditions for which they needed to design a protocol. These conditions were confidentiality, authentication, and integrity. We encouraged them to think aloud while trying to address these conditions and draw on the online whiteboard to help them better explain their thought process.

We started the interview with an explanation of the study and what will be asked of them. We clarified any questions the participant had about the task and the process so far. The task itself, is a protocol design exercise where the participant is asked to design a protocol that satisfies the following conditions:

1. *Confidentiality*: If Alice wants to send a message to Bob, Alice wants to be sure that only Bob can read the message. The message cannot be read by Eve.
2. *Authentication*: If Bob receives a message he expected from Alice, Bob needs to know that Alice did indeed write the message and that it was not written by Mallory pretending to be Alice.
3. *Integrity*: After receiving a message from Alice, Bob needs to be sure that the message has not been tampered with in transmission.

These are the conditions addressed by Public Key Cryptography. Prior to the task, we encouraged the participant to think aloud and draw on the online whiteboard to better explain their thought process. It's important to note that we do not ask the participant what they think Public Key Cryptography is, we do not mention the concept while introducing the task. Instead, we present them with the three conditions that serve as the purpose of Public Key Cryptography, in turn serving as the purpose of their own protocol. Along with the introduction to the characters of Alice and Bob, we offered some additional information in the form of their keypairs. The participant is free to question the purpose of the keypairs or ignore them completely. If the participant asks about the keypairs, we explain that Alice and Bob have a pair of private- and public keys each. However, we do not explain the inverse nature of the keypairs, leaving the definition ambiguous. This primes the participant to question the purpose of the keypairs and possibly discover their inverse relationship, which will in turn help them with the remainder of the task.

Although the conditions are presented in the order of confidentiality, authentication, and integrity, the participant is not required to address them in this order. The participant can work on solving the conditions in an order of their choice. The only limitations set on their thought process will be applied by the logic behind the order of the conditions themselves. If we begin with the condition of authentication instead of confidentiality the scenario is as follows: for Bob to have received a message from Alice, Alice would have to have first sent a message to Bob. This leads to the question ‘How can Alice be sure that the message can only be read by Bob?’, the question associated with the condition of confidentiality. We present confidentiality as the first condition because it shows Alice’s intent, that she intends to send a message to Bob. Authentication and Integrity are not the intentions, they are security measures, additional layers of security. The basal intention is to send a message from Alice to Bob securely, confidentially.

Of course, the order of the conditions is only relevant to the participant if they are using it to try and recall how the process of Public Key Cryptography works, if they are aware of the concept. For those who do not know that the purpose of Public Key Cryptography is to achieve the conditions of confidentiality, authentication, and integrity, they are not bound by their ability to recall. This is even more true for participants who are unaware of the workings of keypairs and do not even question their role in the task.

The participants showed very different approaches to the design process, which will be analysed in the findings but the interaction between the participant and us can be generalised as an thinker-challenger relationship. For example, the participant might think of, suggest an, or recall an idea, and we would challenge that idea by questioning how it works and how it achieves the condition they are trying to address. This relationship could be seen between Rivest, Shamir, and Addleman when trying to conceptualise the theory behind the RSA algorithm. Being the authority on number theory, Addleman would test the practicality of the theories presented by Rivest and Shamir [44]. The intention behind questioning the participants idea was two-fold. Firstly, we want to encourage the participant to elaborate and explore their own idea to see the extent to which they can think. However, we had to ensure that the participant would not wildly digress from the task at hand, this being our second reason. It was important to encourage novel design protocols, because it showcases their mental models, their misconceptions, and similarities to existing mental models seen through the history of Public Key Cryptography. We also challenged their use of vocabulary, especially whenever they introduced a cryptographic term into their design protocol. There were some clear benefits of this, the first being that we could assess whether the participant had correctly explained the purpose of the terminology they had used. Secondly, the participant could clarify the meaning of the term for themselves and decide whether it was the right tool for the task or not.

The task would end once the participant reached a protocol that satisfied the conditions of confidentiality, authentication, and integrity either through a correct explanation of the Public Key Cryptography protocol, or their own novel protocol. Another scenario resulting in the end of the task would be if the participant followed their own novel idea to its extent but came to the realisation that they could not achieve the three conditions with their protocol.

The exit interview allowed the participant to reflect on the task and discuss any challenges or ideas they wanted to talk about. We sent them an exit questionnaire to help summarise their thoughts about the entire study. Following the return of the exit questionnaire, we sent a £20 amazon voucher to the participant.

4.2.3 Data Analysis

We transcribed the audio recordings of the interviews from Zoom.us. The transcription was performed by a GDPR-compliant third-party transcription service approved by the University of Bristol. We also analysed the video recording of the design process shown through the online whiteboard during the diagramming exercise.

4.2.3.1 Transcription Analysis

The purpose of analysing the transcripts is to elicit the participant's mental model and identify factors that guide their thought process and influence their mental models. We adopt the straussian grounded theory to perform our analysis [155].

Open Coding

Through open coding, we reviewed the transcripts and started developing a code book that grew as we as we discovered codes. A code signified a range of events such as the the participant's *initial thought* or when they use a *metaphor* to support their mental model or even when the participant knows about Public Key Cryptography and attempts to present the correct solution, there were cases when some participants were able to *easily recall* what they had learned in the past, while other participants were *trying to remember* but struggling to do so. We also make a note of interactions between the participant and us during the interview. There are instances where participants *clear misconceptions* they think they have before attempting a condition. In other instances the participant is confident that their thought process is correct and when challenged by us, is willing to *defend their idea*.

As we reviewed each transcript, we found that more instances of the code appeared. This was more true for some codes than for others. The instances were saved as quotes from the transcript to support the validity of the code. For each transcript reviewed, we recorded the number of codes identified. For each code, we noted every instances where that code was seen and applied. During the open coding process, we coded precisely and kept revisiting earlier instances of that code before assigning the code to a new instances to ensure that we were applying the code consistently. Strauss & Corbin also recommend that the coder minimise any assumptions they have before the open coding process to help them be more open minded to details that could be missed otherwise [155]. The number of code discoveries decreased as we progressed through the 20 transcripts because instances of the existing codes were becoming more common. We performed the open coding process a total of three times to help refine each version of the code book. By the end of the first open coding round, we had a large set of codes defined and supported by an even larger set of instances in the form of quotations. Before we began the second iteration of the open coding process, we performed a merge of codes within the first code book. A merge would commonly take place between two codes, one that has a large set of supporting instances and another with a much smaller supporting set. The two codes would have a very similar definition along with similar supporting instances. With the refined code book, we performed the second iteration of the open coding process. This time, we used our refined code book when reviewing the transcripts again but were also open to discovering new codes as well. We repeated this process for the third iteration as well.

Axial Coding

The purpose of axial coding is to provoke the idea of relationships between categories and the properties that make them, but the process should not be used rigidly as it could prevent us from "capturing the dynamic flow of the events and the complex nature of the relationships." [104, 155]

We organised the open codes, from the end of the open coding process, into categories. To help with this process, Strauss & Corbin offer a *Coding Paradigm* that defines six subcategories:

1. *Phenomenon*: The phenomenon is the core idea of the category. We identify the phenomenon by observing common experiences shared by the participants from the code book developed during the open coding process.
2. *Causal Conditions*: What caused this phenomenon? We look into the remaining open codes to identify the ones that caused the phenomenon.
3. *Strategies*: Here, we study the actions taken by the participant following the phenomenon.
4. *Consequences*: The outcome of these strategies are the consequences.
5. *Context*: The context is the study itself, the participant has been asked to design a protocol to satisfy the conditions of confidentiality, authentication, and integrity. They have been introduced to the characters of Alice and Bob, and they have been presented with Alice and Bob's keypairs.
6. *Intervening Conditions*: Similar to context, intervening conditions are more general and describe the characteristics of the participants, and how these can influence the strategies. The general characteristics of the participant being that during the recruitment stage, we wanted to recruit participants with some foundational knowledge of cryptography.

Strauss & Corbin explain how these subcategories can relate to each other and form a category. A *causal condition* leads to a *phenomenon*. The participant acts on the phenomenon by developing *strategies* that lead to *consequences*, while this whole process is prefaced with some *context* influenced by *intervening conditions* [155].

Selective Coding

The final step is selective coding. Once we have defined a set of categories through the axial coding process, we define one core category that ties all these categories together. The core category is based on how the categories relate to each other and becomes the basis for the final theory.

4.2.3.2 Diagram Analysis

To encourage the developer's thought process we asked the developer to illustrate their thought process as they are solving the problem. The diagram exercise effectively logs the how the developer visualises a solution to the problem and how they adapt their solution to challenges and questioning by us to help prime to developer's thought process. The questions and challenges we raise help track the evolution of the diagram from initial thoughts to a final attempt.

We compare the diagrams to correct implementation of Public Key Cryptography to distinguish between developers who arrived at the correct implementation and negative cases, where developers did not come to the correct implementation. For negative cases, we analyse the diagram to see at which stages the developers started to feel challenged and deviated from the correct solution. We use these points of deviation to support the analysis of the transcriptions. Negative cases could also show developers who felt confident about their approach and found a different solution for the conditions of Public Key Cryptography.

4.3 Findings

In this section, we present our grounded theory that shows that the participants had predefined mental repositories of threat to look out for and consider when designing a system for achieving the conditions of Public Key Cryptography, and mental models to be invoked to help mitigate these challenges. We observed how participants were able to reflect on their design through the use of an *Attacker Mindset*, identifying potential threats, which in turn results in the participants updating their *Developers' Proposed Solutions* with the aim of improving the security of their design. When their design ideas are challenged and are asked to explain them in more detail, the participants migrated towards mental models that were based on *The Inverse Nature of Encryption & Decryption* and *Concepts Based on Authentication*. Later on in the findings, we will take a closer look at the mental models behind the solutions proposed by the participants to solve for the conditions of *confidentiality*, *authentication*, and *integrity*.

4.3.1 Demographics

Before presenting the grounded theory analysis of the protocol design task, we discuss the experience of the participants below, based on the initial questionnaire they filled prior to the interview. The initial questionnaire asked the following questions:

Participants' experience with cryptographic concepts and programming

There were 13 out of 20 participants who claimed to be *somewhat knowledgeable* about cryptography concepts, such as encryption, digital signatures, and message digests, with almost an even split between seven IT professionals and six students. The difference is more noticeable between the two sub-groups when looking at the seven participants who claimed to be *Knowledgeable* in the concepts of cryptography, with six IT Professionals and one student (Table 4.1).

Level of Knowledge	IT Professional	Student	Total
Not knowledgeable	0	0	0
Somewhat knowledgeable	7	6	13
Knowledgeable	5	2	7
Very knowledgeable	0	0	0

Table 4.1: Count of participants for levels of knowledge in Cryptography. Split show between software developers from Industry and students enrolled in Cryptography and C programming modules.

Of the 20 participants in our study group, 12 participants worked as a professional in the IT Industry. Three of these participants were blockchain developers, with one of them working as a full-stack blockchain developer who develops using ReactJS for the front-end development, Golang for the server, and HyperLedger Fabric for the blockchain platform. One of the IT professionals worked as a software testing specialist in a well-known IT company. Five of these participants worked as software engineers ranging from freelance development to working in senior positions. One of these participants had more than 15 years of experience in network software development in C/Linux and TCL/TK. Out of the 20 participants, only two claimed to have more than 11+ years of experience with C programming. These two participants were among these five software engineers. The remaining four IT professionals consisted of two IT consultants and two software developers. The remaining eight participants of the study group were students who were introduced to C programming as part of their course (Table 4.2).

How often do you use cryptography in your software applications?

The four participants who had never used cryptography in their software applications consisted of two students, a software engineer, and a trainee blockchain developer. The trainee blockchain developer clarified that although they had not had any practical experience with applying cryptography, they had the concept of working. The majority of the study group said that they rarely apply cryptography to their software applications. Out of the nine who fell under this subset, five were students who learned cryptography as part of their course. The participants who claimed to use cryptography occasionally and frequently, were all IT Professionals who were quite familiar with performing security tasks as part of their job (Table 4.3).

Years of experience	IT Professional	Student	Total
< 1 year	7	4	11
1 - 2 years	1	3	4
3 - 5 years	2	0	2
6 - 10 years	1	0	1
11+ years	2	0	2

Table 4.2: Count of participants for years of experience with C programming. Split show between software developers from Industry and students enrolled in Cryptography and C programming modules.

Use of cryptography in software development	IT Professional	Student	Total
Never	2	2	4
Rarely	4	5	9
Occasionally	5	0	5
Frequently	2	0	2

Table 4.3: Count shown for how often the participants apply cryptography to their software applications. Split show between software developers from Industry and students enrolled in Cryptography and C programming modules.

What kind of cryptography-related tasks do you usually implement in your applications?

In reference to Table 4.1, of the 13 participants who claimed to be somewhat knowledgeable, the seven IT professionals primarily performed the following tasks; securing passwords, data encryption, implementing authentication, and generating keys. However, in the case of securing passwords, signs of misconceptions are already starting to appear based on the terminology used by the participants.

> "Encrypt passwords, credit card numbers in business applications." Participant 16 (IT Consultant)

> "Hash user credentials and store in db." Participant 18 (Senior Software Engineer)

The two participants quoted above have made the mistake of confusing the purposes of encryption and hashing. These are two terms that are frequently used in the naming conventions of cryptographic APIs and the supporting documentation. Hashing is used during the process of storing passwords in a database and as an integrity check to see whether or not data has been tampered with during transmission. Hashing algorithms would not be used when attempting to encrypt data as a strong hashing algorithm is a one-way function that outputs a digest that is computationally infeasible to reverse. Whereas the purpose of encryption/decryption functions are to 'lock' and 'unlock' data when required by an authorised user.

Many of the IT professionals, who claim to be somewhat knowledgeable, perform tasks surrounding the authentication process of an application. These participants have experience with software such as OAuth, Open ID Connect, and creating JWT tokens. More cryptographically challenging tasks include the implementation of TLS/SSL certificates with OpenSSL for servers. Libsodium was also mentioned as a cryptographic library that is used for these security tasks. Two of the IT professionals showed experience with generating keypairs and SSH keys between servers and also client-server connections.

Participants also wrote about tasks where encryption is needed to secure user information in servers and databases.

> "Server RSA - Encrypt the rest api response using server rsa key." Participant 8 (Blockchain Developer)

Students of cryptography and C programming, who claimed to be somewhat knowledgeable about the concepts of cryptography, performed tasks primarily based around encryption but have also been introduced to applications of steganography and more historical methods such as the use of morse code.

> "AES encryption for strings or txt files. Steganography for string or text data in image data." Participant 2 (Student)

> "Emulating malicious traffic – sending example c2c commands crypted into morse code." Participant 9 (Student)

One of these students was undertaking a degree apprenticeship, which means that along with being taught the theory behind cryptographic concepts, they had also been exposed to the practical side through working as an apprentice in a work environment as seen through the types of tasks they have performed:

> "Hashing - Use hashing for creating signatures for digital forensics. Implementing Google OAuth - Generating keys manually. Use of cryptographically secure PRNGs." Participant 10 (Cyber Security Degree Apprentice)

Although they did not have practical experience, a few of the student participants showed their understanding of the importance and purpose of cryptographic concepts in real-world applications, which is important to note as we later task them with designing a protocol with the purpose of satisfying a set of conditions.

> "I assume as we are being part of the digital world, majority people do uses mobile net banking or internet banking which surely involves cryptography for the authentication and security.

> Similarly, for digital signature authentication, storing passwords and data cryptography is highly used.
>
> I haven't worked specifically on any cryptographic related task. But I know the basics and would like to join this opportunity to explore this more. Though I am knowledgeable about some of the encryption algorithms used in the background such as RSA, AES, MAC, DSA, RNGs, etc." Participant 13 (MSc Student)

When talking about the tasks performed by participants who claimed to be somewhat knowledgeable about cryptography (participants with a vague idea about various areas of cryptography and what they are used for), seven out of the 13 participants were IT professionals and the remaining six were students. This split of participants varies considerably when looking at the ones who claimed to be knowledgeable about cryptography (participants who are familiar with various areas of cryptography and what they are used for). Out of the seven participants to make this claim, five were IT professionals whereas only two was a student.

The increased level of complexity is noticeable in the security tasks described by the five IT professionals. For example, one of the participants expresses the mindset of an adversary, an attacker whose role it is to perform man-in-the-middle attacks to get the plaintext being transferred through a secure channel discreetly.

> "I have been designing, developing and working in the maintenance of a tls proxy software which acts as a legitimate man-in-the-middle in every tls connections. So, you can assume that the software I write can extract the plaintext that is flowing through the encrypted tls channel but without the knowledge of the actual client and server. I conceptually know the things happening behind the scene in the browsers and in the web servers during tls handshakes and during the bulk encrypted data transfer. I do not know the underlying Mathematics though. I have been using OpenSSL APIs for last ten years. So, I have seen how OpenSSL library has evolved during past 10 years. I have a fair bit of understanding of ssl-3/tls-1.0/tls-1.1/tls-1.2/tls-1.3 rfcs. I have some experience of using openssl tools (s_client, s_server etc). Also have some understanding of the OpenSSL code." Participant 6 (Software Engineer)

Some of the other IT professional participants showed their experience with performing encryption of user information between client-server connections.

> "I encrypt login information from client to server and back. I am encrypting client personal data information requests and responses to and from the application server respectively. I am encrypting application configuration information in the config file." Participant 17 (Software Developer)

After the participant returned the initial questionnaire, we reviewed their responses and, based on suitability for the study, scheduled an interview with them through Zoom.us.

4.3.2 An Introduction to the Categories Elicited from the Straussian Grounded Theory Analysis

Through our grounded theory analysis, we identified a total of 217 open codes, and by observing how these codes related to each other as a result of the axial coding stage, we identified four main categories; *Developers' Proposed Solutions*, *Attacker Mindset*, *The Inverse Nature of Encryption & Decryption*, and *Concepts Based on Authentication*.

4.3.2.1 Developers' Proposed Solutions

The *Developers' Proposed Solutions* category observes the evolution of the participants' proposed solutions from high-level ideas gradually becoming more detailed through the interview process, to the point where we see the mental models behind the proposed solutions. The observation begins with the participant becoming familiar with the conditions of the task, clarifying what exactly is being asked of them, and presenting some initial thoughts. These initial thoughts are high-level ideas such as *Establishing Trust*, *Over the Network*, and *Establishing a Secure Connection*. Often, at this stage, we would challenge the participant to explore and further elaborate on their idea. In response, the participant adopts an *attacker mindset* to identify potential threats associated with the three conditions, resulting in a more specific set of solutions proposed in response to these threats (Figure 4.1).

The participants go on to further detail their proposed solutions, allowing us to elicit their mental models of their proposed solutions, which will be later discussed in Section 4.3.4: *Developers' Mental Models and Misconceptions of Public Key Cryptography*.

4.3.2.2 Attacker Mindset

The *Attacker Mindset* acts as a catalyst, encouraging the participant to think about the solution they have proposed and explain the inner workings of their solution. By adopting the mindset of an attacker, the participant views their proposed solutions through a different perspective. The participant tests their solution and finds ways to break the supporting concepts down. The *Attacker Mindset* also reveals the threats participants associate with the conditions of *confidentiality*, *authentication*, and *integrity* (Figure 4.2).

4.3.2.3 The Inverse Nature of Encryption & Decryption

The *Inverse Nature of Encryption & Decryption* category accounts for solutions that are based off this inverse nature. Participants understood that *Encryption* or an *Encryption Service* was required to address the conditions and when asked to elaborate, concepts such as *Public Key Cryptography*, *RSA*, *Two-Way TLS*, *E2EE* were raised. However, there were many cases where *Symmetric Key Cryptography* and by extension a *Substitution Cipher* was brought up, only for the participant to drop the idea when asked about how the process of distributing the symmetric key would work. At this point the participant would usually suggest an idea based on the *Inverse Nature of Encryption & Decryption*. These ideas would

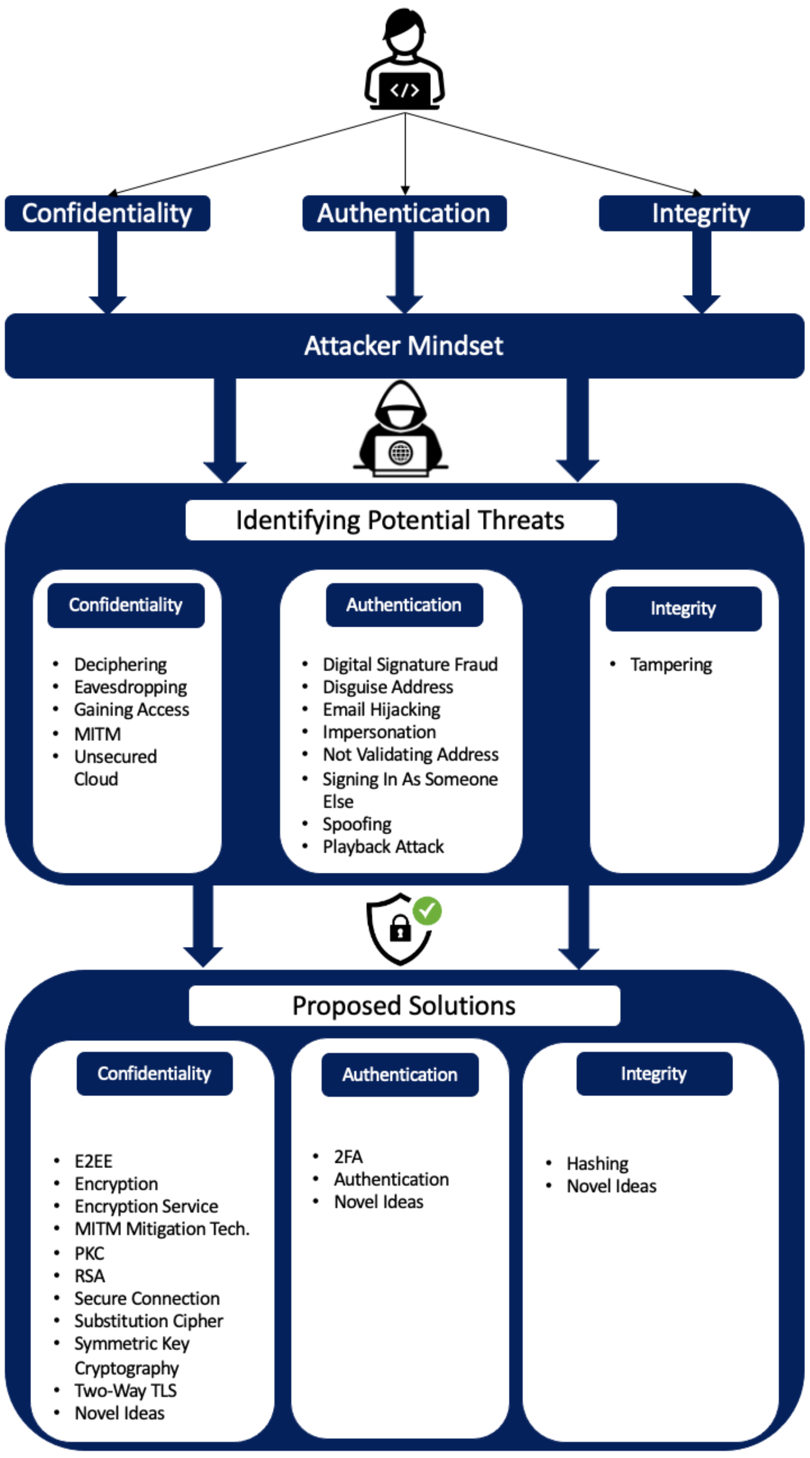


Figure 4.1: Potential Threats and Associated Solutions Proposed by Condition

include ones such as secret & public keys, questions & answers, letters exchanged through the postal system, and many more.

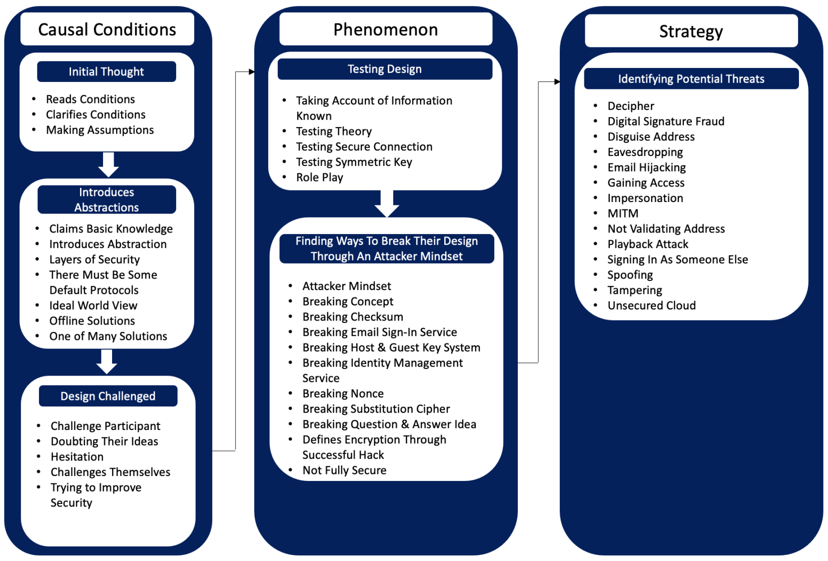


Figure 4.2: Using Attacker Mindset to Break Concept and Identify Potential Threats to Proposed Design

4.3.2.4 Concepts Based on Authentication

This category captured the relationship amongst open codes that were identified while the participant was attempting to solve for the condition of *authentication*. The solutions proposed fell under concepts such as *Two-Factor Authentication*, *Certificate Authorities*, and a number of solutions rooted in misconceptions.

4.3.3 The Grounded Theory: Using Security Goals To Test, Update and Improve the Developers' Mental Repository of Threats and Proposed Solutions

During the process of designing a system to address the security goals of Public Key Cryptography, the participants presented a mental model that aimed to ultimately improve and update a repository of potential threats and proposed solutions they held through an iterative process of using the repository of threats to break the proposed solutions with the aims of making the solutions secure and updating them with the necessary modifications. The process of axial coding concluded with the discovery of four categories; *The Attacker Mindset*, *Developers' Proposed Solutions*, *The Inverse Nature of Encryption & Decryption*, and *Concepts Based on Authentication*. In this section we present a detailed explanation of these categories and how they were defined using Strauss & Corbin's *Coding Paradigm*. We also explore how the four categories relate to each other and form our ultimate grounded theory from the selective coding phase [155]. A full depiction of how the categories relate to each other can be seen in the Appendix (Figure A)

Initial Thoughts & Challenging Developer Mental Models

When first introduced to the task the participant develops their initial thought based on proposed solutions to further build upon. These initial versions of proposed solutions, found in the *phenomenon* sub-category of the category *Developers' Proposed Solutions*, are often presented as a high-level concept such as the need for establishing trust, establishing a mutual TLS connection, or the concept of handshaking. This initial proposed solution serves as a high-level layer of security. We challenge this high-level idea by playing the role of an attacker such as Eve or Mallory, which in turn caused some participants to respond by defending their idea or, more predominantly, hesitate and begin doubting their idea. Here we encourage them to start challenging their own idea while trying to improve the security of their design. This process serves as the *causal conditions* for the *phenomenon* that is the *attacker mindset*.

> "And samewise, likewise, when Bob receives something, so they both need to trust each other, and I think it's probably through a two-way TLS, I feel, I would establish a mutual TLS connection between them." Participant 19 (Software Developer, *Code: Establishing Trust/A Mutual TLS Connection*)

> "So, this is my initial thought, and I might think about something a bit better. I don't know if this is the correct approach." Participant 1 (Student, *Code: Doubting Their Idea*)

Testing the Initial Design & Considering Potential Threats

The participant adopts the *Attacker Mindset* to help identify general threats to consider before detailing the inner workings of how their high-level ideas should be designed. Identifying these general threats for consideration serve as the *strategy* of the *Attacker Mindset*. These general threats include the following:

- Threat - Eavesdropping

  > "Now a secure connection to Bob means first the connection has to identify that it is Bob, and then we have to make the connection secure, so that whatever will pass through that is within that secure area, where nobody else can sort of eavesdrop and pick up the message." Participant 17 (Software Developer, *Code: Threat - Eavesdropping*)

- Threat - Impersonation

  > "So, in this case, Alice cannot be 100 percent sure, "Well, even though the person is saying he's Bob, how can I be 100 percent sure that Bob himself is saying this and not somebody else pretending to be him?"" Participant 1 (Student, *Code: Threat - Impersonation*)

  > "Okay, so there is somebody called Mallory also there somewhere pretending to be Alice." Participant 16 (IT Consultant, *Code: Threat - Impersonation*)

- Threat - Gaining Access

> "This is my initial thought. However, I'm not really certain, but I think that this is not fully secure, as I think Eve could also gain access" Participant 1 (Student, *Threat - Gaining Access*)

- Threat - Tampering

> "Okay. So, what if Alice shares the public key with Bob, but encrypts it with her private key as well so that even if Mallory tries to tamper it, she cannot tamper the message, can she?" Participant 16 (IT Consultant, *Code: Threat - Tampering*)

> "If the message was tampered with, you would know that straight away, first of all, when you were actually decrypting it, you would know, I think, the message is tampered." Participant 19 (Software Developer, *Code: Threat - Tampering*)

As a *consequence* of testing the solutions initially proposed and identifying general threats to consider, the participant takes some time to think and relies on their experience either as a student of cryptography or as software developer. Cryptography students tried to remember cryptographic solutions taught to them to mitigate these general threats, whereas because software developers had more of experience with applying security to applications, they were able to easily recall solutions to help mitigate these threats.

- Solution - Secure Connection

> "So, the first thing that comes to mind is that fact that we need to have a secure connection between Alice and Bob." Participant 1 (Student, *Code: Solution - Secure Connection*)

> "So we have a concept of keys, there are two kind of keys we have, one is the symmetric keys, another one is asymmetric keys to establish this kind of secure connection." Participant 8 (Blockchain Developer, *Code: Solution - Secure Connection*)

- Solution - Encryption Service

> "We need to have some kind of encryption service between both parties." Participant 1 (Student, *Code: Solution - Encryption Service*)

- Solution - Encryption

> "So if Alice wants to send a message to Bob, I would expect her to use some sort of encryption standard." Participant 10 (Cyber Security Degree Apprentice, *Code: Solution - Encryption*)

> "So, here it says if Alice wants to send a message to Bob. Let's say Alice, he sent a message to Bob and Alice wants to show that only Bob can read the message, the message is going to – so it should be encrypted, the data which is going from Alice to Bob, it should be encrypted and we need to make sure that the location where we save it." Participant 14 (Student, *Code: Solution - Encryption*)

- Solution - Hashing

> "So basically like the message is in two parts, you've got the actual message and then you've got like a check sum sort of hash or something that means that if the message is changed slightly it has a totally different hash and it can't be like changed in a couple of ways that cancel out each other and mean that the like check sum is the same. I don't know if I'm using the right terminology but hopefully it makes some sense." Participant 2 (Student, *Code: Solution - Hashing*)

> "So if I am creating a hash means after encrypting all the things if I am creating one encrypted data, if anyone tamper with the data so that data will not be readable, the data will not be decryptable so no-one can tamper that data because it is not readable to anyone else because it is already encrypted such a way that key is not available except this Alice and Bob." Participant 8 (Blockchain Developer, *Code: Solution - Hashing*)

Establishing A Set of Strategies to Help Mitigate General Threats

The *consequences* of the *Attacker Mindset* cause the participant to invoke *strategies* that they have learned either through university or experience in industry and have proposed as the solutions to address the security goals of the task. These *strategies* fall under the category of *Developers' Proposed Solutions*. During the axial coding phase, we managed to further group the solutions based on whether they addressed the conditions of; *confidentiality*, *authentication*, and *integrity*. These *strategies* should be seen as a repository within which the participant holds a series of threats associated with the security goals and the corresponding set of solutions that can be invoked to help mitigate these identified threats.

Now that we have defined the *Attacker Mindset* and introduced some of the *strategies* shown through the *Developers' Proposed Solutions*, we will explain how these two categories are tested, updated and improved through *The Inverse Nature of Encryption & Decryption* and *Concepts Based on Authentication*.

Testing the High-Level Solutions & Considering New Threats

By this stage the participant has offered a series of high-level solutions, invoked from the *strategies* from the *Developers' Proposed Solutions*, to address high-level threats identified and considered using the *Attacker Mindset*. To further explore the participants understanding of how these high-level solutions work, we adopted the *Attacker Mindset* and challenged their proposed solutions even more.

The exploratory process lead to participants identifying more potential threats such as an unsecured cloud and the ability for an attacker to decipher a package sent between Alice and Bob, causing the participant to update their mental repository of threats, as seen in the *strategies* of the *Attacker Mindset*. This update, within the participant's *Attacker Mindset*, acts as a *causal condition* in the *Developers' Proposed Solutions* category. This trigger resulted in the participant doubting their idea and realising that their theory may not be fully secure. They could find that they are able to easily recall a solution for this newly identified threat or they may show hesitation to the point where they become completely stuck, unable to come up with a solution. In situations like this, we encourage their thought process by rephrasing the security goal and ask how the newly identified threat impacts the security of their system. Eventually the participant goes through the process of challenging themselves again, trying even harder to remember solutions that could help mitigate these newly identified threats.

Revising Initial Proposed High-Level Solutions

These *causal conditions* lead the participant to revise their initially proposed solutions of establishing a secure channel in light of identifying these new threats to their system. The participant builds on their initially proposed solutions by further detailing their diagrams introducing more layers of security into their system design. These improved solutions can be seen in the *out of the box solutions* proposed by the participant. For example, the participant may look to delegate a number of security tasks to different mechanism, by justifying that outsourcing these security tasks to reliable third parties could improve the overall security of the system.

> "Alice may have to encrypt it, I think, you know, in Microsoft Outlook, I think there's a way to just have receipts that it has actually been read to Bob." Participant 20 (Student, *Code: Reliable Third Party Improves Security*)

Others attempt to solve for the threats by themselves by using cryptographic APIs. Some participants suggested the use of cryptographic APIs because they knew that cryptographic APIs offer default protocols for the security goals of Public Key Cryptography. Others said that they only engaged with cryptographic APIs when performing security tasks and so they naturally suggested the idea of generating keypairs using a cryptographic API to build and improve on the initially proposed solution of a secure connection.

> "Interviewer: Okay. So how do they agree on a Secret Key? Participant: [Hesitation] There may be some protocols like some default..."
>
> "Yeah, there may be some common thing that Alice and Bob will be in that common group so they will be knowing that what is that protocol and what kind of Secret Key they will use while decrypting the message." Participant 13 (MSc Student, *Code: There Must Be Some Default Protocol*)

> "So essentially what I know of is both need to actually have their own Private Keys, but because it's one-way communication at the moment, Alice will generate the Public and Private. . . .Private Key and Public Key, using any cryptographic API." Participant 19 (Software Developer, *Code: Generating Keypair Using A Cryptographic API*)

This prompts the participant to update their mental repository of initially proposed solutions as part of their *strategies* that fall within the *Developers' Proposed Solutions* category. These newer solutions showed another layer of security in the system's design through solutions such as:

- Solution - Secure Connection
- Solution - Two Way TLS
- Solution - End-To-End Encryption (E2EE)
- Solution - Encryption Service
- Solution - Encryption
- Solution - Man-In-The-Middle (MITM) Mitigation Technologies
- Solution - Public Key Cryptography
- Solution - RSA Algorithm
- Solution - Symmetric Key Cryptography
- Solution - Substitution Cipher
- Solution - Hashing

Testing Revised Solutions & Defining Fundamental Rules for the System Design

The *consequences* of proposing these revised solutions in response to the newly identified threats is that the participant had to test these solutions in the system design which could only be done after explaining these solutions in a lot of detail. To achieve this level of explanation, the participant utilised several techniques found in the *strategies* defined under *The Inverse Nature of Encryption & Decryption* category. These techniques are seen through the participant's ability to reason, define an ideal world view where the system exists in isolation to adverse effects such as the introduction of an attacker, where only Alice and Bob are the only two users of the system. Conversely, some participants actively insert attackers such as Eve and Mallory into the system and used their *Attacker Mindset* while playing the role of the attacker to see the system from their perspective. As a result the participants began to define some rules fundamental to the success of their system's design. For example, many participants said that the private key is generated first and from the private key, the public key is generated.

> "Interviewer: Well what's the Private Key? What's the. . . ? What's the purpose of the Private Key?

> Participant: I think the purpose of the Private Key is to… because I know in Cryptography, like, in order to create a Public Key, you need to have a Private Key, but I forgot about why.” Participant 12 (Software Developer, *Code: Public Key Is Generated From The Private Key*)

The concept of digital signatures was commonly associated with the keypair in the design. Some students of cryptography gave mathematical explanations of cryptographic concepts such as the RSA algorithm, while others explained the details of their high-level solutions at a binary level. Some participants propose *symmetric key cryptography* as a viable solution for solving the security goals of the task. More specifically, these participants present some variation of a substitution cipher, for example the caesar cipher, explained through the example of converting a plaintext message like ‘HELLO’ into a ciphertext representation ‘KHOOR’ by shifting the letters in the alphabet by 3 letters, making 3 the key. However, these participants were shortly faced with the challenge of how to distribute this key between Alice and Bob, which lead them back to square one.

> “Yeah, so I said that, um, so my knowledge in cryptography is very rudimentary, but I know that with the keys, they do have a type of signature that is generated, but I’m not sure for which key it is.” Participant 12 (Software Developer, *Code: Associates Digital Signatures With Keys*)

> “Interviewer: So, if Alice wants to send a message ‘hello’… (Pause) Okay?
>
> Participant: Yeah. Before sending the message, both of them should be aware of the fact that, you know, you are doing it for three letters or so-and-so letters. We have to make sure Bob knows how to decrypt the message before he receives it, because if he doesn’t know that when he receives the message, he just cannot decrypt it.” Participant 15 (Student, *Code: HELLO Caesar Cipher Example*)

Providing Detailed Mental Models to Address the Security Goals of PKC

Based on *The Inverse Nature of Encryption & Decryption* the participants devise a series of mental models that solve for the security goals of *confidentiality*, *authentication*, and in some instances even for *integrity*. Through our axial coding phase, we have organised these mental models after seeing which security goals they address and grouped them further based on commonalities highlighted below (Figure 4.3). These mental models will be explained in more detail later and can be found within the *consequences* subcategory that falls under the *Developers’ Proposed Solutions* category.

Reflecting on Mental Models Based on the Inverse Nature of Encryption & Decryption

The participant has a decision to make here, after reflecting on their detailed mental models of their proposed solutions, they decide whether the system they have designed meets the security goals and and are they confident that their system is secure. If so, they express that they are quite sure that their

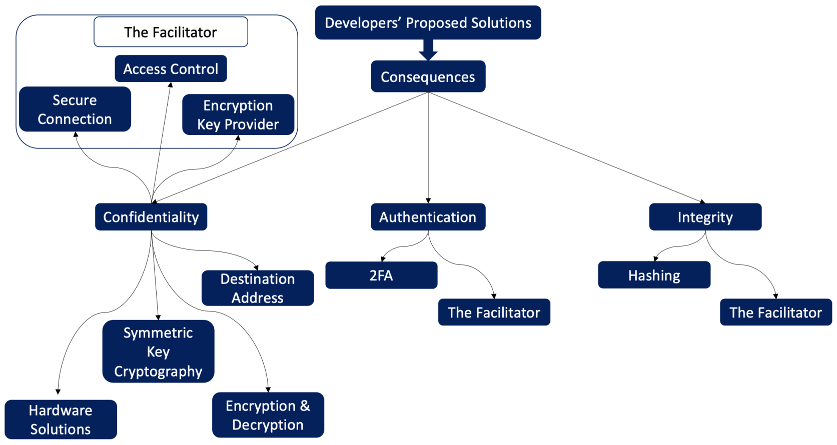


Figure 4.3: Mental Models of Proposed Solutions: Summary of Consequences

solution is secure, they update their mental repository of proposed solutions, and the task concludes. However, if the participant expresses doubts about their system and thinks that their system is not fully secure, this acts as a *causal condition* for the *Attacker Mindset* again so and the cycle repeats with the aim of trying to improve security and achieving the security goals of the task.

The *Concepts Based on Authentication* category resulted from a participants being focused on the challenge of proving their authenticity. The solutions were designed towards solving the security condition of *authentication*.

The *phenomenon* is *caused* in a similar way to the category of *The Inverse Nature of Encryption & Decryption*, where the participant reads the conditions and clarifies their understanding of the task and forms an initial thought about how they should design the system. We challenge the initial thought which leads to the participant doubting the security of their idea and challenging themselves to improve their design. As part of this process of challenging their initial thought, the participant thinks of ways in which their initial thought for the design is vulnerable to potential threats. It is at this stage where the participant refers to their mental repository of threats to list threats associated with the authentication aspect of a secure system.

- Threat - Digital Signature Fraud

> "I don't think. I'm just remembering this from work really because I think someone did challenge, saying a digital signed document claims to be coming from a particular person or company, it doesn't necessarily mean that it actually did, so a malicious individual could masquerade as the sender by producing their own public and private key pair and need these to produce their lovely own digital signature." Participant 20 (Student, *Code: Threat - Digital Signature Fraud*)

- Threat - Disguise Address

"Yeah? So, say, for example, Alice is sending an email to Bob, so alice@gmail.com, right? Now, that's the from address. Eve, what can do, I believe, is Eve can send an email from eve@gmail.com, but can disguise the address with a description saying alice@gmail.com. So, somebody who is not – what to say – like careful enough, can look at the description and say, "Oh, this is from Alice," and open the email, right? But if you look at the details of that email, the description will be like eve@gmail.com, but like the actual value of the eve@gmail.com, but the description will read as alice@gmail.com, right?." Participant 17 (Software Developer, *Code: Threat - Disguise Address*)

- Threat - Email Hijacking

"Now, if Bob gives me a wrong address by any chance, or the email got hijacked or something, then whoever is picking up the email will not have the key again, because I have given it separately to Bob offline, like in a separate way, so they won't be able to read the message." Participant 17 (Software Developer, *Code: Threat - Email Hijacking*)

- Threat - Impersonation

"Yes, I think this would be better, because let's say Alice and Bob have never met. Alice has never seen Bob's signature. Alice has no information basically regarding Bob. How can she be sure that somebody else has signed the paper, or somebody else could be pretending to be Bob." Participant 1 (Student, *Code: Threat - Impersonation*)

- Threat - Signing In As Someone Else

"Let's say Bob is using WhatsApp, now Bob wants to hack Alice's WhatsApp … what he can do is he can just unlink his Google Drive and he can link Alice's Google Drive, if he gets the access, and if he is able to do that then he will have the access to all the messages." Participant 14 (Student, *Code: Threat - Signing In As Someone Else*)

- Threat - Spoofing

"So, I think digital signature is the best way, based on the fact that if Bob says to Alice, "Look, I am Bob, it's indeed me," there are ways in which Eve could create a packet, which spoofs the address and pretends to be somebody else." Participant 1 (Student, *Code: Threat - Spoofing*)

"So, on a real time scenario, so the identity of Alice and Bob – so, I think there are two aspects here. One is the identity and one is the security of the content. So, the security

> of the content can be a certain, because like you are getting a courier from… So, Bob is getting a courier from Alice, from the address which is – I don't know, to be honest, because the address also can be spoofed." Participant 17 (Software Developer, *Code: Threat - Spoofing*)

- Threat - Unsecured Cloud

> "But again, with Google Drive I would say, like, if you are using some sort of drive … I'm not sure if it will be encrypted on that drive or not, but here it should be. It depends upon the app, am I right? Like if there is a feature in the app to encrypt it and decrypt it, it will eventually do i" Participant 14 (Student, *Code: Threat - Unsecured Cloud*)

The *phenomenon* itself was seen when the participants responded to these threats with cryptographic concepts such as digital signatures and a certificate authority that oversees the secure connection between Alice and Bob. The participants also referred to their mental repository of solutions, found in the *strategies* of the *Developers' Proposed Solutions* category, and proposed solutions such as authentication & verification, two-factor authentication (2FA), and end-to-end-encryption (E2EE). When asked about these solutions, the participant provided definitions to help clarify their thoughts. This lead to the participant devising *strategies* for achieving the security goal of *authentication* through a detailed explanation of proposed solutions. These solutions are a *consequence* of the *Developers' Proposed Solutions phenomenon*. We will discuss the mental models behind these solutions collectively at a later point.

During the open coding phase, we discovered a series of codes that capture the behaviors shown by the participant as they are trying to explain the details of their proposed solutions, seen in the *strategy* sub-category under *Concepts Based on Authentication*. Since it was a more detailed explanation of the solutions invoked during the *phenomenon* the participant showed another layer of security for that solution. The participants were confident that their proposed solutions addressed the condition of *authentication* and at times made this explicitly clear. Some participants initially offered a low-level cryptographic explanation of their solution, but when challenged, proceeded to simplify their solution for the task.

The *consequences* of the proposed *strategies* were to test the participants' conceptualisation for vulnerabilities, making sure that the proposed solution addressed the condition of *authentication*, and steps taken in preparation to conclude the task.

At the beginning of the process of testing the participants' conceptualisation for vulnerabilities, many participants showed hesitation to make further changes to their proposed solutions and expressed that, due to them having only a fundamental understanding of cryptographic concepts, their solution may not be the standard way in which the conditions are achieved, unknowingly referring to Public Key Cryptography. In response, we challenged them by asking them to summarise their proposed solutions and ensure that they fully address the security goals of the task. By challenging themselves and allowing themselves to doubt the security of their own solution, the participant began to test their theory by adopting the *Attacker Mindset* and starting the cycle again, fueled by the aim of trying to improve the security of their proposed solutions. The participant role played as the attackers seen in Eve and Mallory to try and break their system. If vulnerabilities were found, the identified threats stored in the participants mental repository would be updated, found under the *strategy* sub-category of the *Attacker Mindset* category. This update of identified threats would in turn cause the participant to re-evaluate and update their mental repository of solutions associated with these newly identified threats, found under the *strategies* of the *Developers' Proposed Solutions* category. However, if no vulnerabilities are found from the theory testing process, the participant can move on to making sure that their proposed solutions addresses the security conditions of Public Key Cryptography. Here the participant clarifies the condition and explains their proposed solutions through a series of events or through an example, while looking for validation to confirm that they have achieved the condition of *authentication*. While preparing to conclude the task, the participant reflects on their solution and may reason as to why they delegated some of the processes of their solutions to third parties, stating that the use of reliable third parties could improve the security of the system, and that it was one of many solutions that could have resulted from the design process. Finally, the participant summarised their solution and concluded that they are quite sure that it is secure and ends the task.

4.3.4 Developers' Mental Models and Misconceptions of Public Key Cryptography

In this section we explore the mental models held by developers as they try to design a system that satisfies the conditions addressed by Public Key Cryptography. These mental models are represented in the grounded theory as the *consequences* invoked from the *Developers' Proposed Solutions* category. Through the analysis of their mental models, we present a detailed explanation of the proposed solutions and show how they address the conditions of *confidentiality*, *authentication*, and *integrity*, if they do address them at all. We also highlight the areas where their mental models differ from the standard way in which Public Key Cryptography works.

Through our grounded theory analysis, we have elicited 11 mental models. Amongst these 11 mental models there are specific models like *The Facilitator* that have been applied by the participants to address all three conditions of Public Key Cryptography. On the other hand, we elicited other mental models that only addressed one of the three conditions, for example, designs that fell under the group *Two-Factor Authentication* only addressed the condition of *authentication*. We will discuss each of the 11 mental models below.

4.3.4.1 Mental Models for Confidentiality

The first three mental models that we will introduce could be grouped under *The Facilitator*. The Facilitator is a system between Alice and Bob that facilitates the process of establishing a secure line of communication between the two parties. Many of the participants have integrated the facilitator's purpose with mental models shown in their design process for confidentiality. The Facilitator is seen again when we look at the mental models surrounding *authentication* and *integrity*.

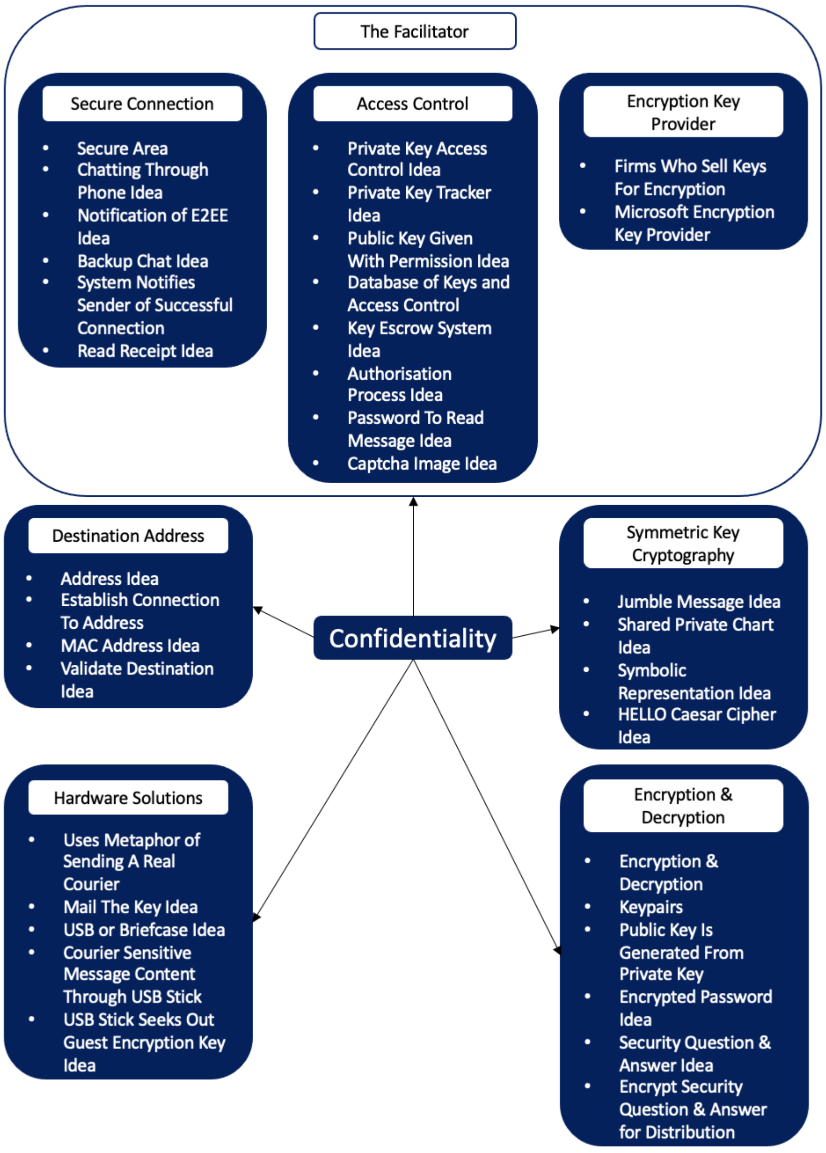

The Facilitator
Secure Connection
Secure Area
Chatting Through Phone Idea
Notification of E2EE Idea
Backup Chat Idea
System Notifies Sender of Successful Connection
Read Receipt Idea
Access Control
Private Key Access Control Idea
Private Key Tracker Idea
Public Key Given With Permission Idea
Database of Keys and Access Control
Key Escrow System Idea
Authorisation Process Idea
Password To Read Message Idea
Captcha Image Idea
Encryption Key Provider
Firms Who Sell Keys For Encryption
Microsoft Encryption Key Provider
Destination Address
Address Idea
Establish Connection To Address
MAC Address Idea
Validate Destination Idea
Confidentiality
Symmetric Key Cryptography
Jumble Message Idea
Shared Private Chart Idea
Symbolic Representation Idea
HELLO Caesar Cipher Idea
Hardware Solutions
Uses Metaphor of Sending A Real Courier
Mail The Key Idea
USB or Briefcase Idea
Courier Sensitive Message Content Through USB Stick
USB Stick Seeks Out Guest Encryption Key Idea
Encryption & Decryption
Encryption & Decryption
Keypairs
Public Key Is Generated From Private Key
Encrypted Password Idea
Security Question & Answer Idea
Encrypt Security Question & Answer for Distribution

Figure 4.4: Mental Models of Proposed Solutions: Detailed Consequences (Confidentiality)

The Facilitator

Model #1: Secure Connection

After reading the conditions of the task, the study group as a whole displayed an understanding that a secure connection was required to achieve the conditions of *confidentiality*, *authentication*, and *integrity*. Participants stressed the need for a *secure area* in which Alice and Bob can safely communicate. The general idea was that this *secure area* would be established by a *system* that notified both Alice and Bob that the secure connection was established.

Participant 14 takes inspiration from real-life applications such as WhatsApp and suggests that Alice and Bob should use a 'Platform' that offers End-to-End Encryption (E2EE). The platform should

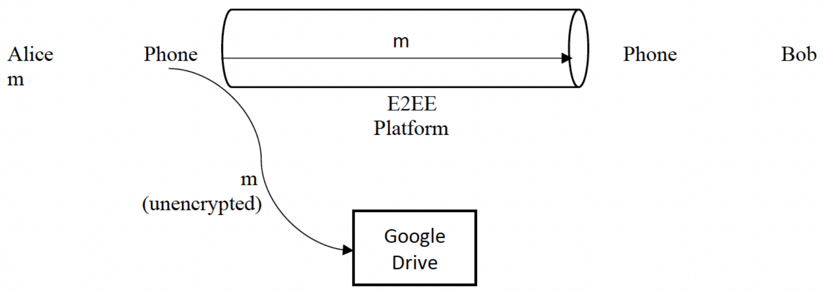


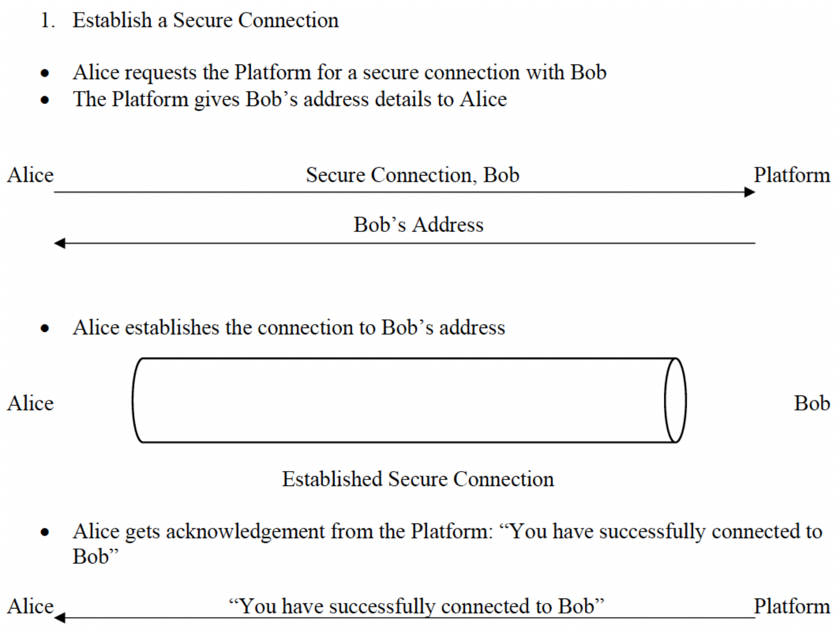
Figure 4.5: Participant 14: E2EE Channel and a vulnerable connection to Google Drive

provide the option to encrypt the messages and to take backups of the chats to store in the user's Google Drive account. Participant 14 explains a potential vulnerability in the design. The participant explains that although the connection between Alice and Bob is secured through End-to-End Encryption, the connection between WhatsApp and Google Drive may not be. This means that an adversary, like Eve, can sync their own WhatsApp account to Alice's Google Drive and read the messages in plaintext. The vulnerability described by the participant would also indicate that there is no authentication system in place between Alice's WhatsApp account and Alice's Google Drive account, because Eve can sync her phone to Alice's Google Drive account with ease, by the participant's description. The vulnerability also suggests that Google Drive does not have any encryption mechanisms to ensure that stored chats are secured and not available in plaintext. The participant concludes that if this vulnerability is resolved, the condition of *confidentiality* would be satisfied (Figure 4.5, Figure A).

Participant 17 designs a system for *confidentiality* that is two-fold: point one being to establish a secure connection and point two being the encryption of the message. On request for a secure connection from Alice, the 'Platform' sends Bob's address as a reply. Using this address, Alice can establish a secure connection with Bob and be notified by the platform that the connection has been established successfully (Figure 4.6, Figure A). The participant proposes a different mechanism, separate from the 'Platform', for encrypting the message before sending it through the secure channel. An 'Encryption Key Sharing Provider' distributes a 'Host Key' and a 'Guest Key' to Alice and Bob, respectively. Using the 'Host Key' Alice can encrypt the message which can later be decrypted by Bob using the 'Guest Key' (Figure 4.7). Alice is assured of *confidentiality* by the 'Encryption Key Sharing Provider'. When compared to the standard protocol of Public Key Cryptography, the element of trust shifts from Alice's use of Bob's public key, being certain that only Bob can open it with his private key, to a third-party, a facilitator who guarantees that messages encrypted with the 'Host Key' can only be decrypted with the 'Guest Key' and that the 'Guest Key' is with Bob, in turn assuring Alice of the *confidentiality* of the message (Figure 4.8, Figure A).

Model #2: Access Control

A commonly seen misconception surrounding the nature of public & private keys is that the public key plays the role of the symmetric key, a shared key between both parties with the purpose of encrypting



1. Establish a Secure Connection

- Alice requests the Platform for a secure connection with Bob
- The Platform gives Bob's address details to Alice

Alice
Secure Connection, Bob
Platform
Bob's Address

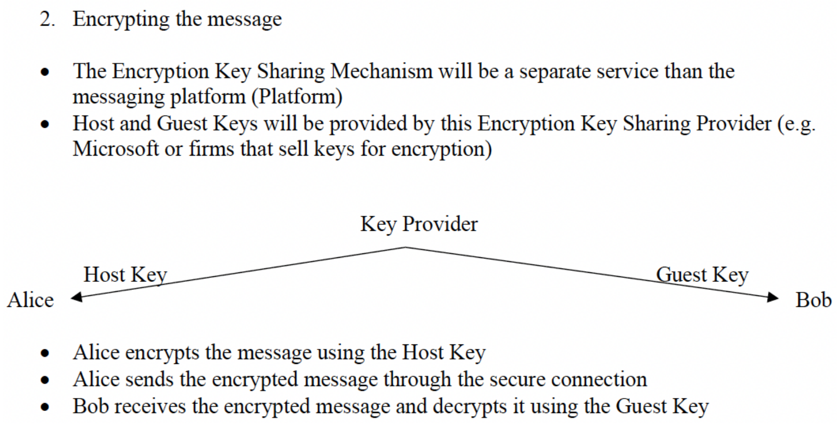
- Alice establishes the connection to Bob's address

Alice
Bob
Established Secure Connection

- Alice gets acknowledgement from the Platform: "You have successfully connected to Bob"

Alice
"You have successfully connected to Bob"
Platform

Figure 4.6: Participant 17: Establishing a Secure Connection



2. Encrypting the message

- The Encryption Key Sharing Mechanism will be a separate service than the messaging platform (Platform)
- Host and Guest Keys will be provided by this Encryption Key Sharing Provider (e.g. Microsoft or firms that sell keys for encryption)

- Alice encrypts the message using the Host Key
- Alice sends the encrypted message through the secure connection
- Bob receives the encrypted message and decrypts it using the Guest Key

Figure 4.7: Participant 17: Distribution of Host & Guest Keys by Key Provider

the message, and the private key plays only the role of the decryption mechanism. This is only half true, as the nature of public & private keys are mutually inverse, where a message encrypted by a public key can be decrypted by a private key and vice versa.

Participant 12 offer a novel understanding of how public & private keys relate to each other. The public key is shared between both parties, Alice and Bob, with the ability to encrypt and decrypt. The public key is generated from the private key and the private key has a list of novel features in Participant 12's design. The private key stores a list of people who have possession of the public key, so if the public key is with Bob, the private key has logged this information. However, if Eve, an adversary, intercepts the public key and gains possession of it, the private key knows and has the ability to cut off the functionality of the public key and stop it from working, for Eve. In the design, explained by

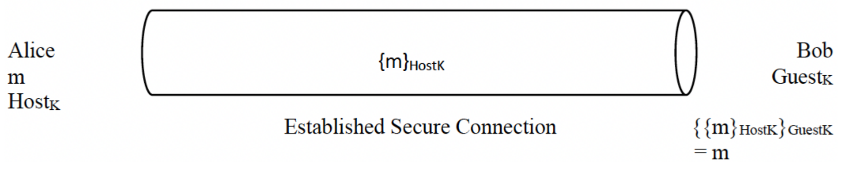
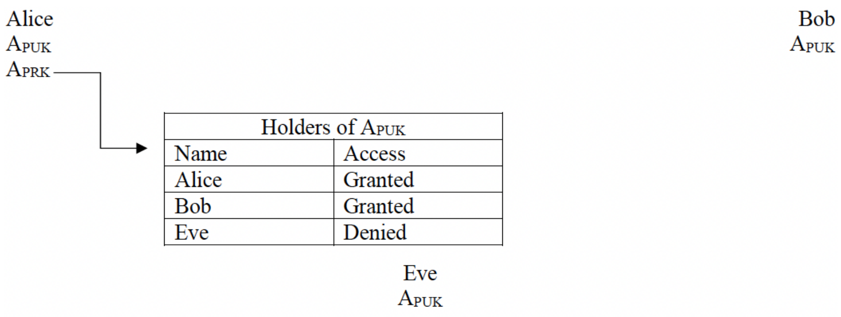


Figure 4.8: Participant 17: Confidentiality Design for Confidentiality



Holders of APUK	
Name	Access
Alice	Granted
Bob	Granted
Eve	Denied

Figure 4.9: Participant 12: Public Key used as Symmetric Key, Private Key used for Access Control

Participant 12, the public key plays the role of a symmetric key while the private key plays the role of an access control system accessible only to one individual, in this case Alice (Figure 4.9, Figure A).

Model #3: Encryption Key Provider

Some participants make a clear distinction between a secure connection and an encrypted message. These designs integrate different mechanisms together, some for establishing a secure connection through a facilitator, while others for encrypting the message itself before it gets sent through the secure channel.

Participant 17 designs a system for *confidentiality* that is two-fold: point one being to establish a secure connection and point two being the encryption of the message. On request for a secure connection from Alice, the ‘Platform’ sends Bob’s address as a reply. Using this address, Alice can establish a secure connection with Bob and be notified by the platform that the connection has been established successfully (Figure 4.6, Figure A). The participant proposes a different mechanism, separate from the ‘Platform’, for encrypting the message before sending it through the secure channel. An ‘Encryption Key Sharing Provider’ distributes a ‘Host Key’ and a ‘Guest Key’ to Alice and Bob, respectively. Using the ‘Host Key’ Alice can encrypt the message which can later be decrypted by Bob using the ‘Guest Key’ (Figure 4.7). Alice is assured of *confidentiality* by the ‘Encryption Key Sharing Provider’. When compared to the standard protocol of Public Key Cryptography, the element of trust shifts from Alice’s use of Bob’s public key, being certain that only Bob can open it with his private key, to a third-party, a facilitator who guarantees that messages encrypted with the ‘Host Key’ can only be decrypted with the ‘Guest Key’ and that the ‘Guest Key’ is with Bob, in turn assuring Alice of the *confidentiality* of the message (Figure 4.8, Figure A).

Model #4: Destination Address

When first introduced to the task, it was common to see participants suggest the idea of establishing a secure connection between Alice and Bob, and when challenged to elaborate on their proposed solution, the participants often cited symmetric key cryptography. Then we would ask the participant how the symmetric key is agreed upon and shared between the two parties. The paradoxical solution they would respond with would be the need for a secure connection to safely share the symmetric key. This proved to be a point where the participant would realise the real challenge of the task and would challenge themselves to think about a solution. Participants 12 and 17 raised the idea of using the recipient's known address by Alice, the sender, as a destination address to which she wants to establish a secure connection. This way, Alice can be sure that she is securely communicating with Bob.

> "So, if it is real time, step one is connecting to Bob, so Bob's address has to be certain. So, this information has to be given to us from Bob, that, "This is my address and, okay, these are my address details."" Participant 17 (Software Developer)

> "Yeah, something to that, something to that extent, something equivalent to a mac address, because I know for a mac address, like, certain... there are, like, I think it's like a 32-bit type of information, because the first four, the first two bits, sorry, first four bits, I think, it is used to identify the manufacturer, so if something can be implemented in this case for Bob, so Alice can identify that as Bob, and then be able to send the key." Participant 12 (Software Developer)

Model #5: Symmetric Key Cryptography

Four of the 20 participants attempted to address the condition of *confidentiality* with some variation of a *symmetric key*. After reading the first condition, Participant 1 identifies that without an 'Encryption Service' the line of communication between Alice and Bob is vulnerable to the threat of eavesdropping. When asked how this encryption service works, the participant explains that it performs symmetric key cryptography, denoted by K_{SYM}. When describing the cryptography behind the symmetric key, the participant recalls concepts taught to them during their university course, a commonality seen with three other participants who are also students and who have also brought up the idea of a symmetric key to address the first condition. Participant 1 explains the workings of a *substitution cipher*, more specifically a *caesar cipher*, through the example of shifting the letters of the word 'HELLO' by three resulting in 'KHOOR'. The number three represents the symmetric key, used to both encrypt and decrypt the message. After being challenged on what would happen to their system if an adversary intercepted the key, the participant introduces the concept of *public and private* keys.

Participant 15 begins with the concept of a substitution cipher, quoting the same example as Participant 1, with the word 'HELLO' being shifted by 3 letter to spell 'KHOOR', the ciphertext. When asked what would happen if Eve, the adversary intercepted the key and the message, the participant explored the idea of using a password (e.g. '123'), known only to Alice and Bob. When asked how

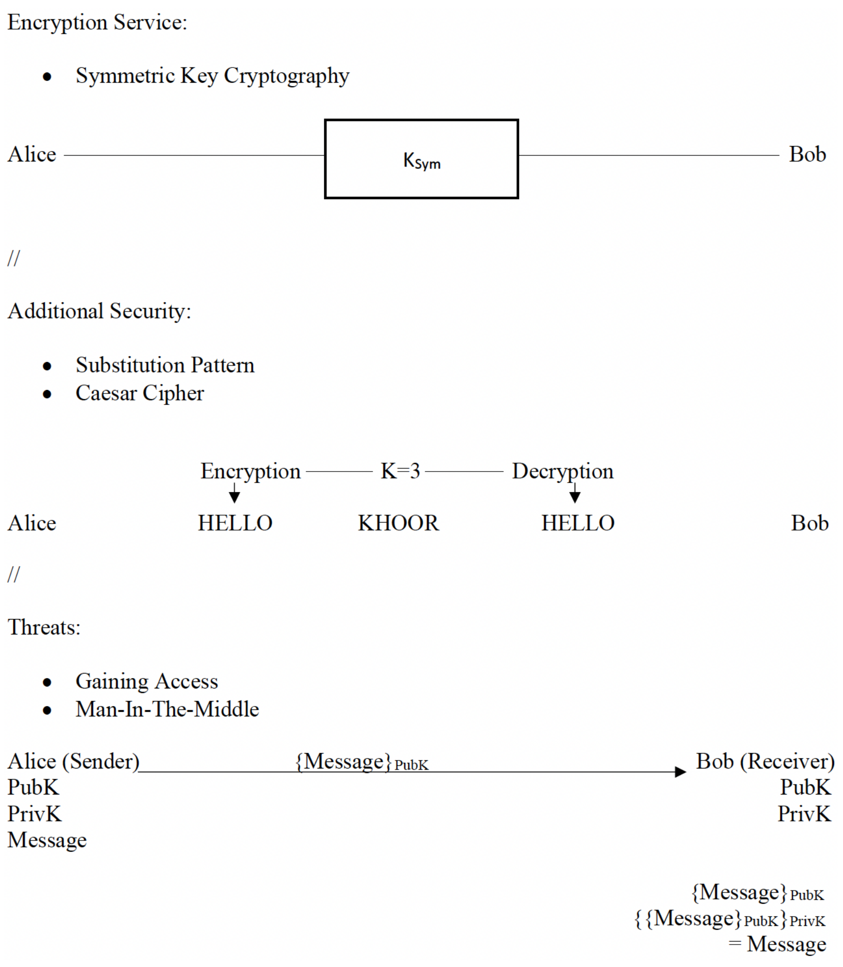
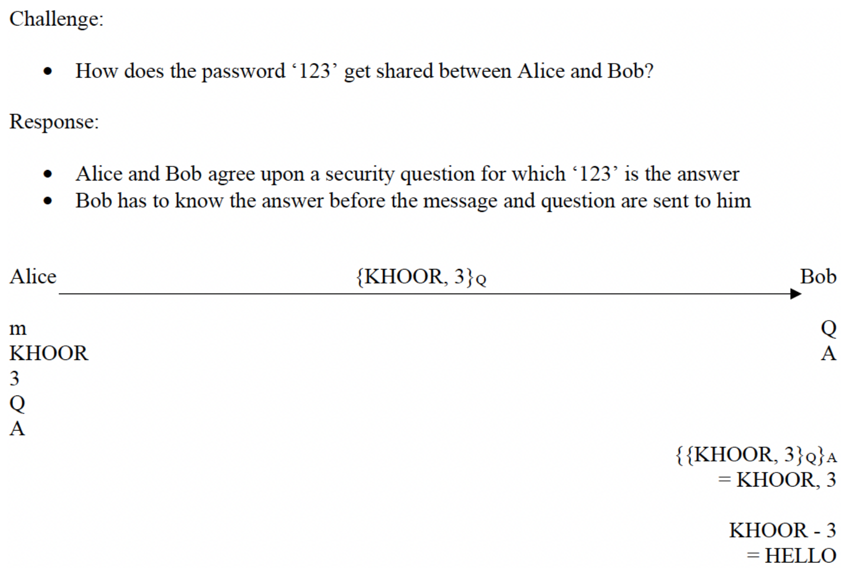


Figure 4.10: Participant 1: Encryption Service

this password is distributed between Alice and Bob, the participant introduces the idea of a *Question & Answer* protocol. Both Alice and Bob agree on a security question for which the password '123' is the answer. This model is quite similar to the lock and key mental model originally used to explain the Public Key Cryptography concept. The question represents the open lock distributed to people Bob wants to communicate with. Alice closes the lock on a message and sends it back to Bob. Bob has the key, represented by the answer '123' to open the lock and read the message. However, the *Question & Answer* protocol is a symmetric protocol because both the *question* and the *answer* are known to both Alice and Bob, so in this protocol the *question* represents the symmetric key's ability to encrypt a message, and the *answer* represents the symmetric key's ability to decrypt a message (Figure 4.11, Figure A).

Model #6: Hardware Solutions

Taking inspiration from real-world message delivery systems, such as the postal system, participants proposed designs that incorporated aspects of these systems to ensure security.


Challenge:

- How does the password ‘123’ get shared between Alice and Bob?

Response:

- Alice and Bob agree upon a security question for which ‘123’ is the answer
- Bob has to know the answer before the message and question are sent to him

Figure 4.11: Participant 15: Q & A System

Participant 11 relates the concept of Symmetric Key Cryptography to the idea of ‘handshaking’. When it comes to describing the distribution of the symmetric key between the two parties, Alice and Bob, the participant suggests the use of a ‘USB Pen’ being handed over from Alice to Bob, or the use of a ‘briefcase’ (Figure A). Similarly, Participant 17 explains that in the case of sending sensitive content, Alice should send a USB stick through a courier to Bob. Along with the encrypted message itself, the USB stick has the ability to authenticate Alice’s identity by some relation to Alice’s ‘Host Key’. Once received, Bob is instructed by a message in the courier package to plug the USB stick into his machine. Once plugged in, the USB stick ‘seeks out’ Bob’s ‘Guest Key’ and uses it to decrypt the message that it carries (Figure A). In Participant 14’s design, Alice is assigned a PIN number when she first registers with the ‘Platform’. From then onwards, whenever Alice wants to start a chat with Bob using the E2EE protocol, her phone requests the PIN number, known only to her. The purpose of this mechanism is to address a theoretical threat identified by the participant. If Alice unlocks her phone and leaves it lying around, Mallory could pick it up and start a conversation with Bob, while pretending to be Alice. The PIN system prevents the threat of impersonation. The participant also explains how this mechanism could be used to authenticate Bob and ensure Alice that the message can only be read after Bob inputs his PIN number into his phone (Figure A).

Model #7: Encryption & Decryption

When attempting to solve for the condition of *confidentiality* 11 out of the 20 participants presented some form of a solution based on the concept of *Encryption & Decryption*, so much so that one of the four main categories to result from the axial coding phase of the grounded theory analysis was based on the *Inverse Nature of Encryption & Decryption*. These participants showed knowledge and application of concepts such as keypairs, both at a standard level where they demonstrate that a message can be encrypted by a public key and decrypted by a private key, and vice versa, but also at lower levels of abstractions where other participants asymmetric algorithms, like RSA, at a mathematical level to explain how exactly the public and private keys are generated and can perform the inverse function of each other (Figure A).

4.3.4.2 Mental Models for Authentication

Model #8: Two-Factor Authentication

Similar to *Hardware Solutions*, there are many examples of real-world applications of authentication that everyday users, including our study group, are familiar with. Some of the participants apply these examples to solve for the condition of *authentication*.

Participant 13 devises the use of a system that authenticates Alice and Bob through hardware devices. These authentication devices allow Alice and Bob to sign-in and authenticate themselves using their a digital login like their 'Gmail account' or a 'UserID/Password' form. The sign-in information and confirmation are relayed between the device and the system that oversees the authentication process (Figure 4.13). The use of emails to authenticate a user is a common feature in real-world applications and the influence of this can be seen in the participants solution for the condition of *authentication* here (Figure 4.13).

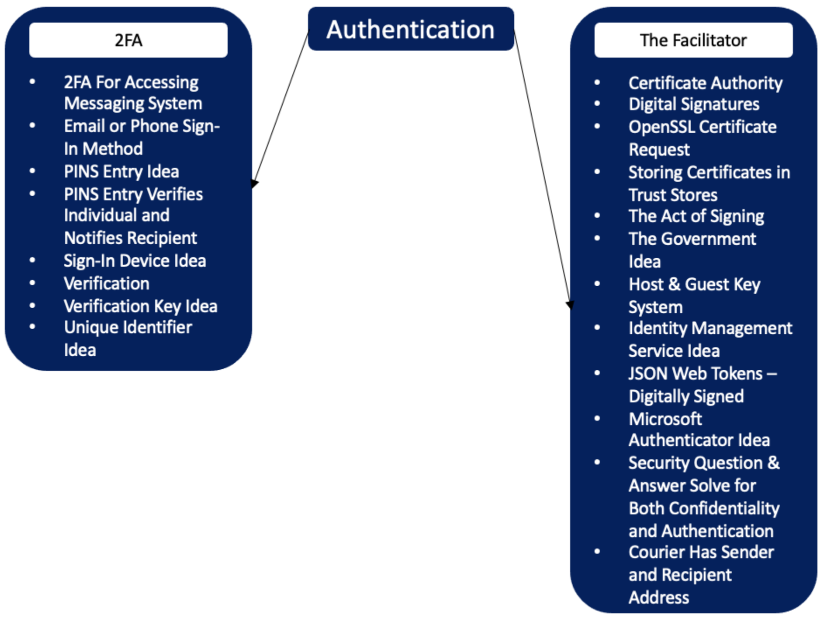

Authentication
2FA
2FA For Accessing Messaging System
Email or Phone Sign-In Method
PINS Entry Idea
PINS Entry Verifies Individual and Notifies Recipient
Sign-In Device Idea
Verification
Verification Key Idea
Unique Identifier Idea
The Facilitator
Certificate Authority
Digital Signatures
OpenSSL Certificate Request
Storing Certificates in Trust Stores
The Act of Signing
The Government Idea
Host & Guest Key System
Identity Management Service Idea
JSON Web Tokens – Digitally Signed
Microsoft Authenticator Idea
Security Question & Answer Solve for Both Confidentiality and Authentication
Courier Has Sender and Recipient Address

Figure 4.12: Mental Models of Proposed Solutions: Detailed Consequences (Authentication)

Participant 14 also refers to real-world applications when explaining the detailed steps of the Two-Factor Authentication process. This concept is brought forward as a means to solve for the condition of *authentication*. Not only is the concept introduced but also explained in a lot of detail by the participant. This can be seen through the steps being relayed between Alice, 'The System', and Bob in Figure 4.14.

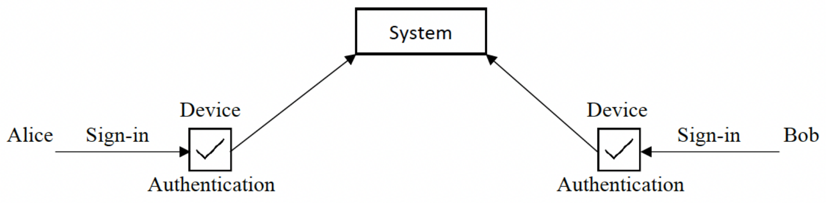


Figure 4.13: Participant 13: Authentication System

Participant 14 recognises that Mallory serves as a potential threat to the system as she can impersonate Alice and chat with Bob, for which the idea of a private PIN number is introduced. The participant explains that if Alice successfully authenticates herself and then leaves her phone lying around, Mallory can pick up the phone and pretend to be Alice (Figure 4.14).

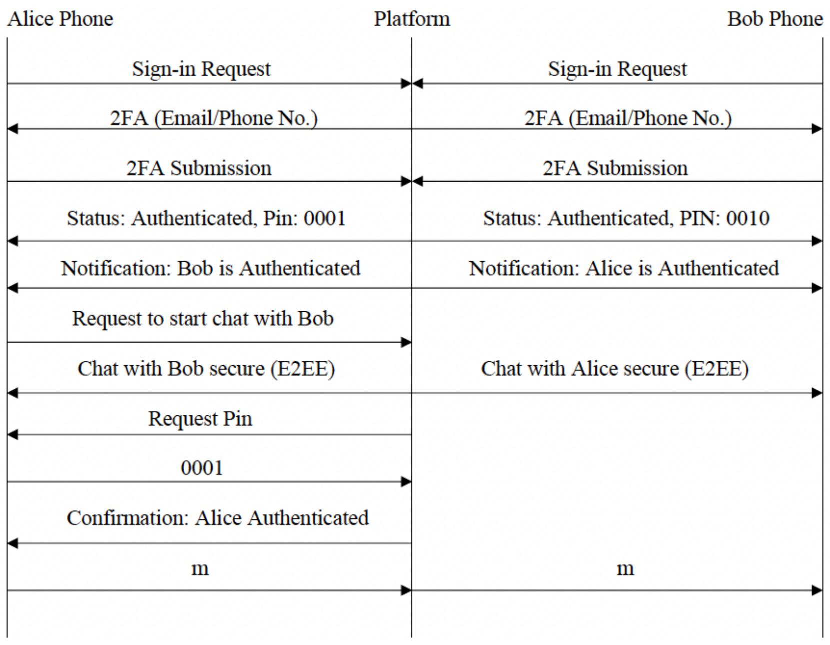
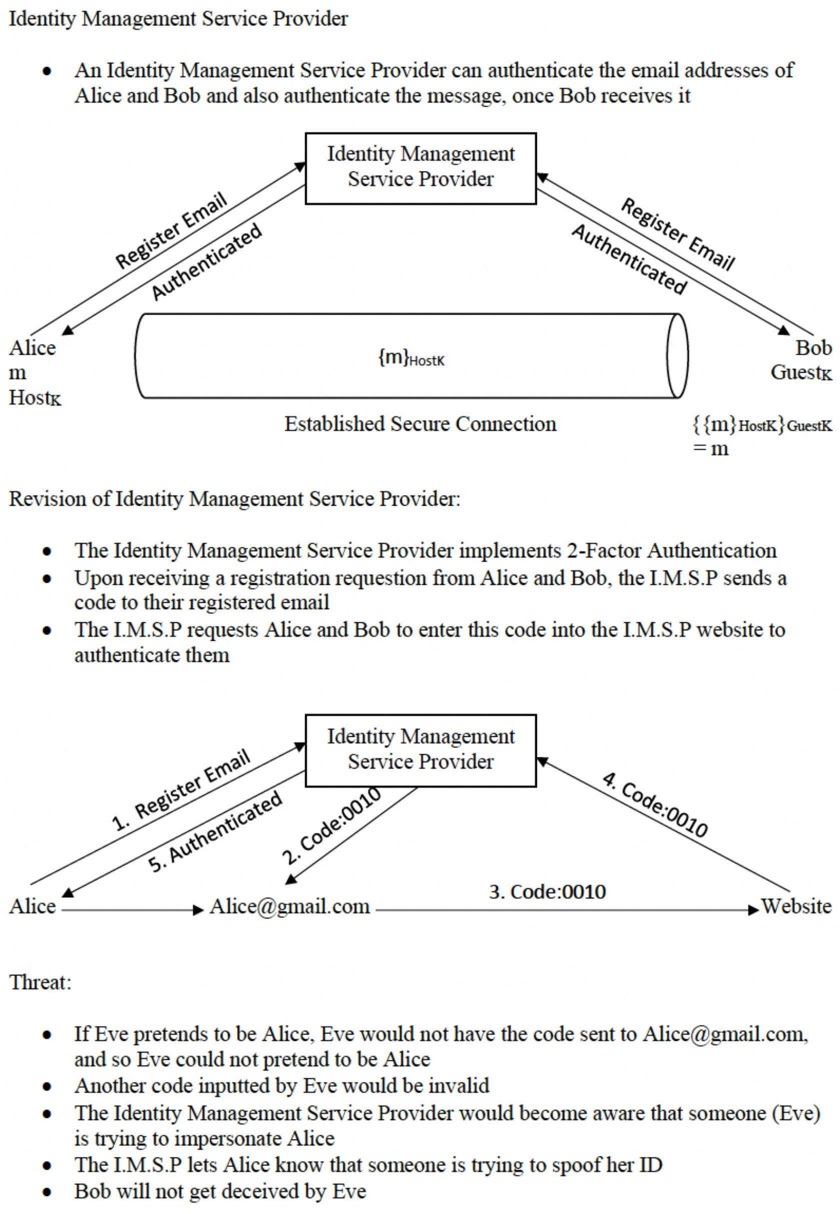

Alice Phone
Platform
Bob Phone
Sign-in Request
Sign-in Request
2FA (Email/Phone No.)
2FA (Email/Phone No.)
2FA Submission
2FA Submission
Status: Authenticated, Pin: 0001
Status: Authenticated, PIN: 0010
Notification: Bob is Authenticated
Notification: Alice is Authenticated
Request to start chat with Bob
Chat with Bob secure (E2EE)
Chat with Alice secure (E2EE)
Request Pin
0001
Confirmation: Alice Authenticated
m
m

Figure 4.14: Participant 14: Two-Factor Authentication Process

The case study of Participant 17's design is interesting because unlike the other participants who have designed one application that handles all of the three conditions of Public Key Cryptography, Participant 17 has delegated many of the responsibilities to many different facilitators. In the case of addressing the condition of *authentication*, the participant introduces the 'Identity Management Service Provider'. The 'Identity Management Service Provider' performs two-factor authentication using the individual's email with which they registered with and by sending that email a code. The 'Identity Management Service Provider' asks Alice and Bob to enter the code into a website designed to verify the code and in turn authenticate their identity. The participant tests the security of this system by running through a scenario where Eve tries to impersonate Alice. The system works so long as Eve cannot access Alice's email account and gain the authentication code (Figure 4.15).


Identity Management Service Provider

- An Identity Management Service Provider can authenticate the email addresses of Alice and Bob and also authenticate the message, once Bob receives it

Identity Management Service Provider
Register Email
Authenticated
Register Email
Authenticated
Alice
m
HostK
{m}HostK
Bob
GuestK
Established Secure Connection
{{m}HostK}GuestK = m

Revision of Identity Management Service Provider:

- The Identity Management Service Provider implements 2-Factor Authentication
- Upon receiving a registration requestion from Alice and Bob, the I.M.S.P sends a code to their registered email
- The I.M.S.P requests Alice and Bob to enter this code into the I.M.S.P website to authenticate them

Threat:

- If Eve pretends to be Alice, Eve would not have the code sent to Alice@gmail.com, and so Eve could not pretend to be Alice
- Another code inputted by Eve would be invalid
- The Identity Management Service Provider would become aware that someone (Eve) is trying to impersonate Alice
- The I.M.S.P lets Alice know that someone is trying to spoof her ID
- Bob will not get deceived by Eve

Figure 4.15: Participant 17: Identity Management Service Provider

Model #9: The Facilitator

The *Facilitator* plays an integral role in the design of many participants as they attempt to solve for *authentication*. Participant 6 showed an understanding of certificate authorities in the process of signing digital signatures and authenticating parties. The participant in referred to as the 'Government' in Participant 6's design (Figure 4.19). Participant 14 suggests a system that uses third party services, specifically the Microsoft Authenticator, to address *authentication*.

> "...and that two factor authentication like verifying your account either through email or some, like, let's say Microsoft Authenticator or some text message or some phone call . . . then it will be more secure and it will help others to rely, like, that message is secured." Participant 14 (Student)

Participant 17 presented the 'Identity Management Service Provider', a facilitator which we previously discussed in the *Two-Factor Authentication* mental model (Figure 4.15).

4.3.4.3 Mental Models for Integrity

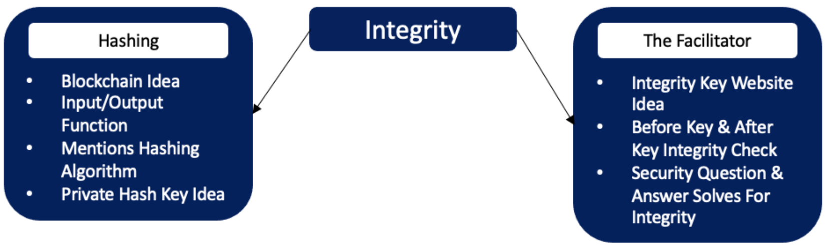
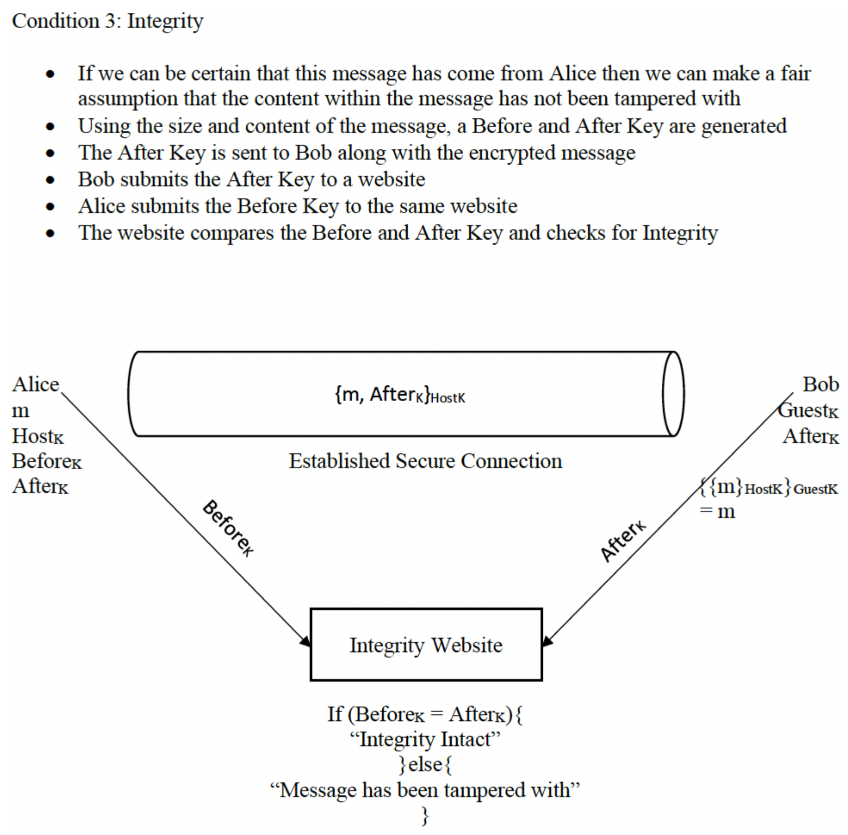

Integrity
Hashing
Blockchain Idea
Input/Output Function
Mentions Hashing Algorithm
Private Hash Key Idea
The Facilitator
Integrity Key Website Idea
Before Key & After Key Integrity Check
Security Question & Answer Solves For Integrity

Figure 4.16: Mental Models of Proposed Solutions: Detailed Consequences (Integrity)

Model #10: Hashing

When solving for the condition of *integrity* the majority of the study group were able to correctly associated the term 'hashing' with the challenge and the idea of 'comparing hashes' to check for integrity. However, when asked to explain their understanding of hashing, only a small number of participants correctly demonstrated 'the standard way'.

> "Alice can make a hash format of this message and send the message on to Bob so when Bob received the message receives an error, Bob can check the hash to the message and that will then. . . Bob will know that the message has not tampered in the middle." Participant 5 (Software Developer)



Condition 3: Integrity

- If we can be certain that this message has come from Alice then we can make a fair assumption that the content within the message has not been tampered with
- Using the size and content of the message, a Before and After Key are generated
- The After Key is sent to Bob along with the encrypted message
- Bob submits the After Key to a website
- Alice submits the Before Key to the same website
- The website compares the Before and After Key and checks for Integrity

```
If (BeforeK = AfterK){
    "Integrity Intact"
}else{
    "Message has been tampered with"
}
```

Figure 4.17: Participant 17: Integrity Check using Before & After Keys

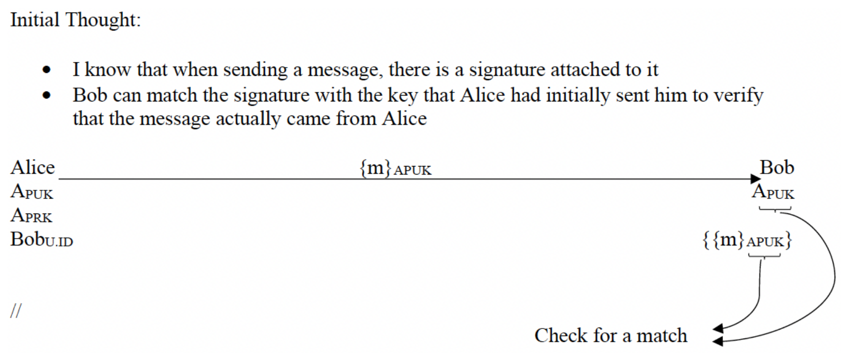
Model #11: The Facilitator

Along with *confidentiality* and *authentication*, the mental model of *the facilitator* is also seen to have a role when it comes to addressing the condition of *integrity*.

Using the size and content of the message, a new facilitator calculates a ‘Before Key’ and an ‘After Key’. Participant 17’s facilitator sends the ‘Before Key’ to Alice and the ‘After Key’ to Bob along with the encrypted message. Both Alice and Bob submit the ‘Before Key’ and ‘After Key’ to a website. The website’s role is to check for integrity by comparing the ‘Before Key’ and ‘After Key’ to check for a match (Figure 4.17).

4.3.4.4 Proposed Solutions Based on Misconceptions

So far, we have introduced a series of mental models associated with the conditions of *confidentiality*, *authentication*, and *integrity*. Many of these mental models are based on the participants experience with real-world applications of security such as *End-to-End Encryption*, *PINSEntry*, or even the *Postal System*. Only a small number of participants were able to correctly rediscover the process of Public Key Cryptography. However, of these participants who were familiar with the nature of public and private keys and their role in Public Key Cryptography, some showed clear misconceptions when it came


Initial Thought:

- I know that when sending a message, there is a signature attached to it
- Bob can match the signature with the key that Alice had initially sent him to verify that the message actually came from Alice

Figure 4.18: Participant 12: Comparison of Alice's Public Key to prove for Authentication

to applying these keys. Relevant terms such as *encryption*, *digital signatures*, and *hashing* were used confidently by the participants when describing their system design but their misconceptions showed when asked to explain their understanding of these concepts and how they apply to the task at hand. Below, we cover a number of misconceptions seen during the task.

Participant 12 solves for the condition of *authentication* based on two misconceptions. The first being that Bob can be sure that the public key he received from Alice did in fact come from Alice not only because he received it but also because it works. In the protocol designed by this participant the private key has an access control feature which can be used to enable and disable a given public key's functionality. The second misconception is that Alice's public key is her digital signature and can be used to prove her authenticity. Once Bob receives a message, encrypted using Alice's public key, Bob compares the key used to encrypt the message to the key he received from Alice during the key exchange process (Figure 4.18).

The general idea of a digital signature seemed to be well understood by many, as a way to authenticate the identity of Alice from a message received by Bob. However, the representation of this concept varied quite a bit between the participants, although there were some similarities that could be identified between them.

In Participant 6's design, Alice concatenates her digital signature to the ciphertext before sending it to Bob, an idea also proposed by Participant 3. This is because the key they use to send messages between Alice and Bob is a shared key, so Alice needs to prove her identity in a way that is different to encrypting the message using her private key and expecting Bob's decryption of the message using her public key to serve as proof that the message came from her. The process of generating the digital signature, however, is an asymmetric process that involves Alice and the certificate authority, which the participant refers to as the 'Government'. Alice sends an application to the 'Government' that consists of her name, her job, and other details that identify her, this serves as her unique ID. The government signs Alice's unique ID using it's private key, a black key, in a similar way that a certificate authority would do so. This signed ID is sent back to Alice who can now use it as her digital signature. The 'Government's' public key, a blue key, is known to all, so if Bob wants to verify Alice's digital signature, Bob can use the blue key on the digital signature to access Alice's unique ID. If successful, Bob knows that Alice's

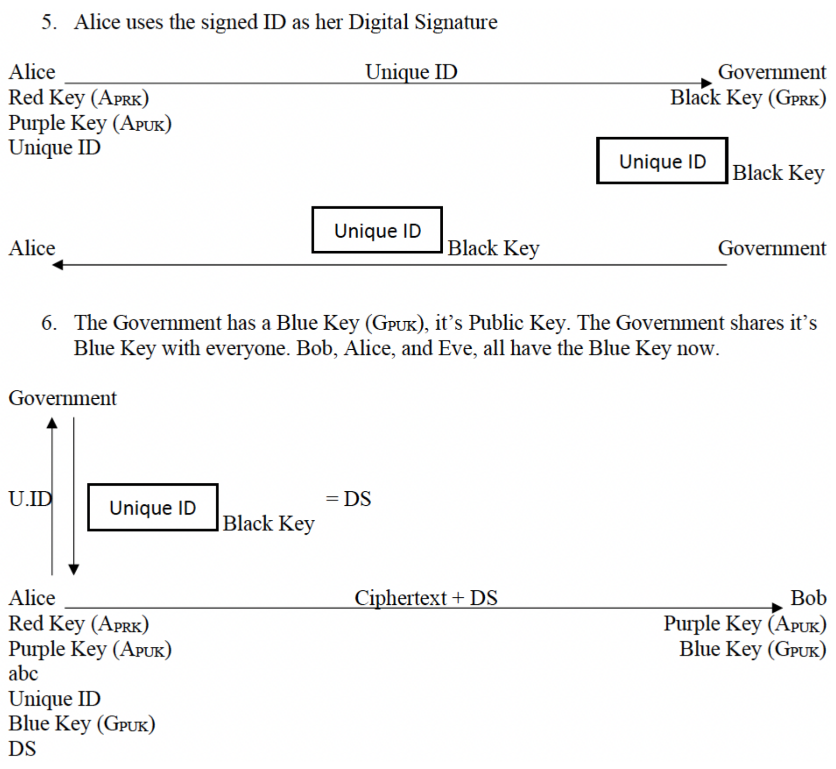
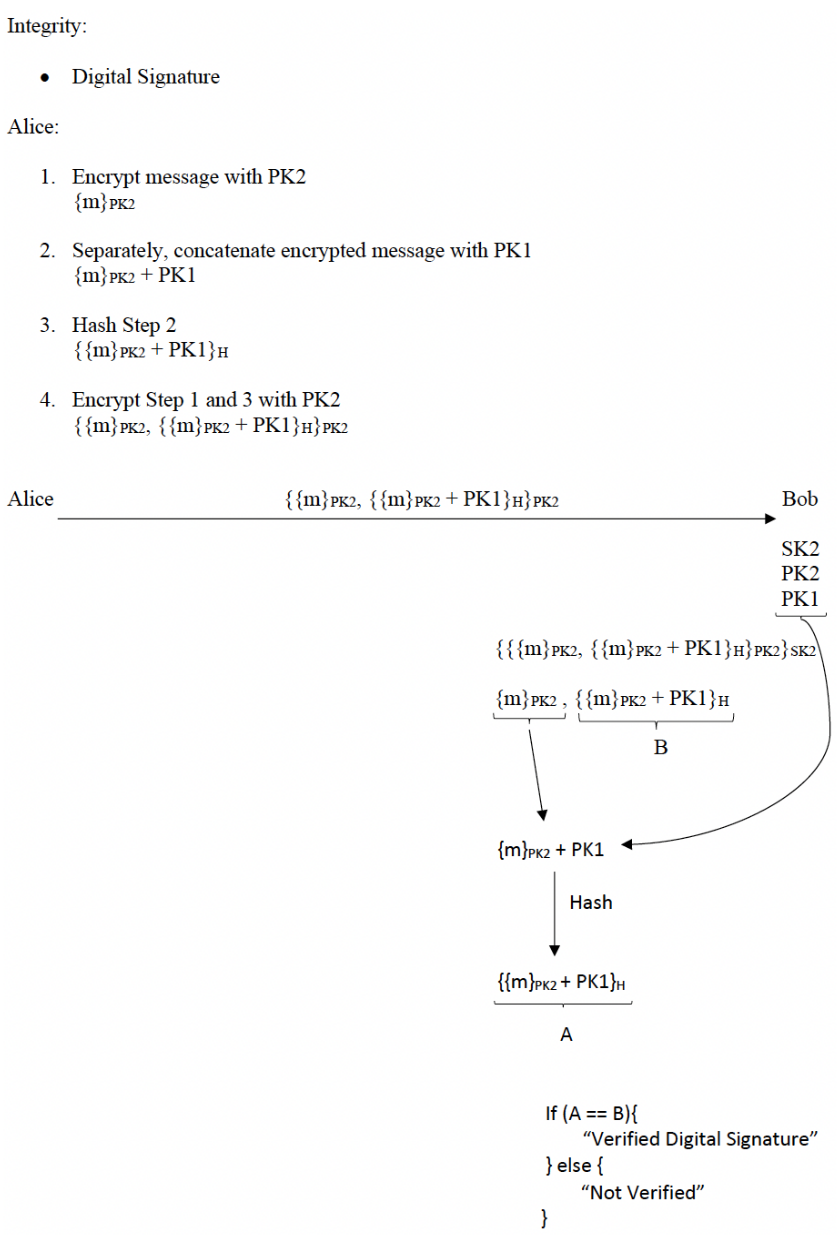


Figure 4.19: Participant 6: Digital Signature with Government Intervention

unique ID was signed by the 'Government' and that Alice is who she says she is (Figure 4.19).

Participant 3 designs a solution that combines concepts associated with the conditions of *authentication* and *integrity*. The participants initial thought being to make use of digital signatures. They do this by concatenating Alice's public key to an encrypted message, encrypted using Bob's public key. This is a misconception seen on the participants part as the public key of any individual does not prove their authenticity as it is available to everyone. Through the standard way, the hash algorithm is applied only to the message, to then later be compared to a hash of the plaintext message by the recipient, Bob. However, in Participant 3's design a hash is taken of the concatenation of the encrypted message and Alice's public key. Once the package is received by Bob, he has all the pieces of information to recreate the content within the hash value and perform the hash on it again. In this system, Bob is tasked with constructing a piece of information that effectively plays the role of the plaintext in the standard way (Figure 4.20).



Integrity:

- Digital Signature

Alice:

1. Encrypt message with PK2
   $\{m\}_{PK2}$
2. Separately, concatenate encrypted message with PK1
   $\{m\}_{PK2} + PK1$
3. Hash Step 2
   $\{\{m\}_{PK2} + PK1\}_H$
4. Encrypt Step 1 and 3 with PK2
   $\{\{m\}_{PK2}, \{\{m\}_{PK2} + PK1\}_H\}_{PK2}$

Alice
$\{\{m\}_{PK2}, \{\{m\}_{PK2} + PK1\}_H\}_{PK2}$
Bob
SK2
PK2
PK1
$\{\{\{m\}_{PK2}, \{\{m\}_{PK2} + PK1\}_H\}_{PK2}\}_{SK2}$
$\underbrace{\{m\}_{PK2}}$, $\underbrace{\{\{m\}_{PK2} + PK1\}_H}_{B}$
$\{m\}_{PK2} + PK1$
Hash
$\underbrace{\{\{m\}_{PK2} + PK1\}_H}_{A}$

```
If (A == B){
      "Verified Digital Signature"
} else {
      "Not Verified"
}
```

Figure 4.20: Participant 3: Unique Integrity Check

4.4 Discussion

4.4.1 The cycle of influence between real-world security applications and mental models

4.4.1.1 Security applications influence on mental models

Many of the mental models elicited from the study were based on real-world security applications that the participants engage with on a day-to-day basis. Applications such as WhatsApp and its use of *End-to-End Encryption* influenced participants when designing for the condition of *confidentiality*. Participant 14 used their experience with WhatsApp as a framework to build upon during the task. During the design process, the participant introduced the concept of a 'Platform' that facilitates the establishment of the secure end-to-end encrypted connection. The participant also builds on the framework by adopting an *attacker mindset* to consider all possible threats, known to them, and suggest appropriate mitigations.

There is a parallel to be noted between secure development practices in organisations and the grounded theory explained in our study. An important aspect of Microsoft's Secure Development Lifecycle is threat modeling. The Microsoft SDL Threat Modeling Tool is comprised of five major steps:

1. *Define Security Requirements*: In our study, this was represented through the framing of the conditions of *confidentiality*, *authentication*, and *integrity*. The basis of the task was a real-world scenario where two parties, Alice and Bob, need to communicate in a secure way.
2. *Create an application diagram*: Along with thinking aloud, the participants found it useful to diagram their system designs to better explain how it worked.
3. *Identifying threats*: Here is where the *attacker mindset* was used by the participants to test the state of their designed system. During this process, the participant would test and, if needed, update their mental repository of threats against their design.
4. *Mitigating threats*: For any threats that were identified, the participant proposed solutions from their mental repository that related to their repository of threats.
5. *Validating that threats have been mitigated*: After applying their proposed solutions to their designs, the participants tested their systems again against their mental repository of threats using an *attacker mindset*. If they were satisfied that the threats were mitigated, the task would end, but if not, they participant would repeat the cycle again.

From this observation, we can see that the way in which the participants performed the task, captured in our grounded theory, closely resembles the security practice carried our through the threat modeling process of Microsoft's Secure Development Lifecycle.

Participant 12 presented a novel understanding of how the public and private keys function and relate to each other. Along with the ability to encrypt and decrypt, the private key also serves as an access control mechanism, with the ability to enable and disable permissions and access to the public key by those who hold it. This form of access control is also seen within organisations who integrate role-based access control into their systems as it addresses many of the security requirements of industrial and government organisations [53]. Employees who fall under a specific role are granted appropriate permissions based on their responsibilities to control their exposure to the system as a whole. Activities such as online banking commonly practice authentication measures such as *two-Factor Authentication* and *PINSEntry*. Examples of *Two-Factor Authentication* such as *PINSEntry*, *Email SignIn*, and *Verification* became recurrent themes seen throughout the interview process. A more physical form of security was seen through the mental models of Participant 11, who recalled the act of signing a letter when addressing the condition of *authentication* (Figure A), and Participant 20, who combined the idea of a phyiscal stamp placed onto an envelope and the condition of *integrity* resulting in a *Digital Stamp Duty*.

The observation to be made here is that these real-world applications have one common factor and that is that they require the participant to engage with them at some level. When it comes to two-factor authentication, for example, the participant has to sometimes create passwords or PINs and memorise them. In organisations, the participants may initially test the boundaries of their role-based permissions. Like these, many of the mental models elicited serve as the abstractions through which developers see security and design secure systems.

4.4.1.2 Examples of Security Applications based on our elicited mental models

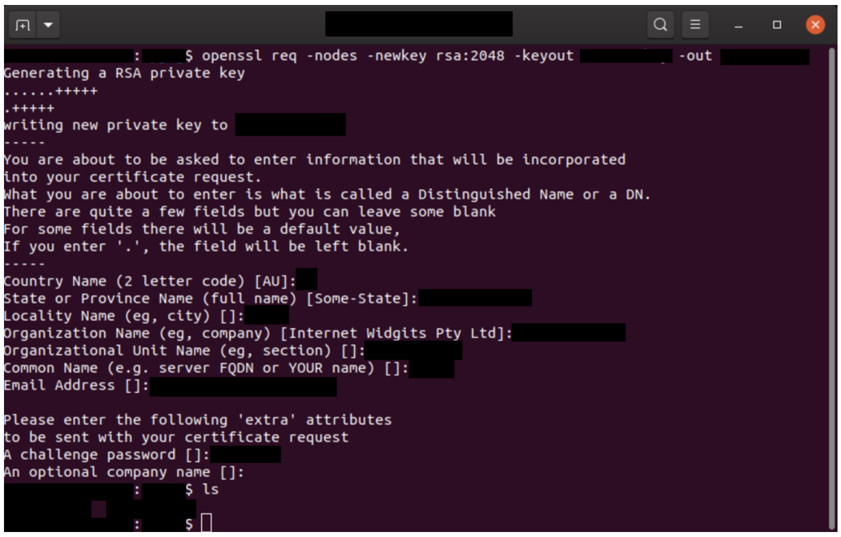
We have discussed the origins of Public Key Cryptography and the role mental models played. There are examples of more recent security protocols and security applications for our mental models. Chen et al. [28] proposed a secure color-code key exchange protocol for mobile chat applications in 2018, a concept seen through Participant 6's design process where they too color-coded the public and private keys of all parties involved in the secure system. Chen's novel protocol describes how session keys could be assigned randomly generated color-codes using simple exclusive-or operations for each session [28]. In 2013, Arvin et al. [9] designed a secure online USB login system that utilised a hybrid-cryptosystem comprised of cryptography and steganography to deliver a low-cost secure online two-factor authentication system. Arvin et al.'s authentication system stored both a password and a security token in a hardware device, a secure USB[9]. Two-Factor authentication was one of the primary mental models shown by participants when it came to addressing the condition of *authentication*. The idea of using a USB was raised by both Participant 11 and Participant 17 when tasked with the condition of *confidentiality*, and this idea fell under the mental model *Hardware Solutions*. Arvin's design relies on many different mechanisms working together, specifically three main components: a website, a hardware device, and a trusted third party[9]. This approach closely resembles the mental model of *The Facilitator* for *authentication*.

4.4.2 Can the developers' mental models for Public Key Cryptography serve as usable abstractions for secure programming?

When we look at the API design of cryptographic libraries, such as OpenSSL, we can see some of the applications of the mental models elicited from our study. OpenSSL presents a high-level cryptographic

API known as the *Envelope API* that uses the abstraction of securing a letter within an envelope in its API design. The process of Public Key Cryptography is shown through two primary functions: *envelope_seal* and *envelope_open*. If Alice wanted to send a message securely to Bob, Alice would write and seal the message using *envelope_seal* and send the message to Bob who could open the message using the *envelope_open* function. Libsodium, another C-based cryptographic library, presents a similar set of functions in *crypto_box* and *crypto_box open*. Both these sets of functions agree with the participants understanding of the *inverse nature of encryption & decryption*.

When creating a private key using the OpenSSL Command Line interface, we see the application of Participant 6's *Unique ID* concept, an identity comprised of the parties' name, email address, postcode etc. The OpenSSL Command Line interface request similar details from the user (Figure 4.21).



[redacted]

```
[redacted]:[redacted]$ openssl req -nodes -newkey rsa:2048 -keyout [redacted] -out [redacted]
Generating a RSA private key
......+++++
.+++++
writing new private key to [redacted]
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:[redacted]
State or Province Name (full name) [Some-State]:[redacted]
Locality Name (eg, city) []:[redacted]
Organization Name (eg, company) [Internet Widgits Pty Ltd]:[redacted]
Organizational Unit Name (eg, section) []:[redacted]
Common Name (e.g. server FQDN or YOUR name) []:[redacted]
Email Address []:[redacted]

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:[redacted]
An optional company name []:
[redacted]:[redacted]$ ls
[redacted]
[redacted]:[redacted]$
```

Figure 4.21: Creating an RSA Private Key using the OpenSSL Command Line Interface

So with similarities such as these being seen in the designs of cryptographic APIs, why do developers still struggle to use them? Our study shows that the cause of this issue begins far before the developer has even begun using the cryptographic APIs. The understanding, held by the developers and seen through their mental models, greatly misaligns with how Public Key Cryptography actually works. This means that, irrespective of how cryptographic libraries present Public Key Cryptography through their APIs, the developer will not be able to correctly use them. For example, the fact that OpenSSL uses the concept of the postal system through their *Envelope API* should be somewhat familiar to some developers. But this is not the case because the functions *envelope_seal* and *envelope_open* require the correct use of parameters, such as Alice's keypair and Bob's keypair, to work. Unless the developer has the correct understanding of Public Key Cryptography, they will struggle to not only use the correct parameters but also to understand how the parameters relate to each other. An argument could be made that documentation can help clear up any misconceptions developers have about how Public Key Cryptography works and this is true to a degree but it places a lot of responsibility on the developers that they may not be aware of. Firstly, unless the developer has gone through a protocol design exercise like in our study, they do not have a clearly defined mental model that they have thought aloud and diagrammed, so their mental model is highly abstracted as seen in the initial set of solutions proposed in the grounded theory (e.g. E2EE, Secure Connection, Secure Area). Secondly, after reading the documentation of a cryptographic API and trying to familiarise themselves with how the functions work and how the parameters relate to each other, they need to mentally adopt the abstraction used in the cryptographic API design, which could mean breaking some of their own mental models.

Green & Smith proposed 10 principles with the aim of improving the usability of security APIs. One of these principles was *Don't break the developer's paradigm*. This principle explains that, when it comes to cryptography, the developer's understanding is often limited to certain paradigms such as the encryption, digital signatures, and other functions. Firstly, the mental models elicited from our study show that there is a great difference between knowing the correct terms to use for a given task and knowing how those terms actually work in practice, for example, the misconceptions seen under the mental model *Access Control* such as the private key being able to track the location of the distributed public key and the private key's ability to enable and disable access of the public key. Secondly, the principle works but only to an extent. A good case study to explain why is the OpenSSL's *Envelope API*. At a very high abstraction level the *Envelope API* follows the principles and may be seen as usable and inviting by the everyday developer. At this point, in terms of the choice of abstraction, there may be a number of developers who share a similar idea of a secure envelope as seen from participants of the study group. However, once the developer engages with the API, the misalignments between their understanding of cryptographic concepts and how cryptographic concepts actually work start to show. At this point, the developer's paradigm needs to break because the mental models they hold and the misconceptions they have are so different from how certain cryptographic concepts work that it will either cause them to misuse it or prevent them from using it all together. Many of the mental models elicited can be used as examples to support this. Participants who bring up concepts such as public & private keys, digital signatures, and hashing, for example, use the term correctly but when asked to explain how the terms work in practice show misconceptions which can lead to cryptographic misuse (E.g. Participant 3: Unique Integrity Check: Figure 4.20). On the other hand, participant who proposed completely novel solutions, unrelated to cryptographic concepts such as public & private keys and digital signatures, may struggle to use the cryptographic API in the first place more so that misuse cryptographic functions. For example, Participant 17 will have the responsibility of either attempting to map their mental model of an *Identity Management Service Provider* (which falls under the *Facilitator* mental model for the condition of *authentication*) to cryptographic concepts like keypairs and digital signatures or drop their mental model entirely, breaking their paradigm, and adopting the abstraction through which these cryptographic concepts are presented in the cryptographic API.

To answer the question, yes, the mental models elicited from our study can be used to design abstractions for secure programming but this does not mean that the abstractions will be usable (Refer to Chapter 3). The mental models elicited from our study cannot help developers with the use of cryptographic APIs for the reasons discussed in this section. While we can't be sure that the mental models can lead to usable abstractions we can takeaway some insights from our study to help inform the secure software development lifecycle. The usable abstractions for secure programming do not necessarily need to take place during the development stage of the secure software development lifecycle. Instead, after studying our participants' thought processes, we can look into the idea of developing usable abstractions earlier, during the secure design stage of the secure software development lifecycle. We should focus on the thought process, captured through our grounded theory analysis, and how factors such as *attacker mindset* help the developer refine not only their thinking but also their design. The key finding from our study is the shift in objective from trying to design a usable abstraction to help developers with secure programming to focusing on applying the developers' thought process to initial steps towards improving the usability of cryptographic APIs, we discuss this in more detail in Chapter 5 when we explain our conceptual shift-left initiative: CryptoBridge.

A valuable insight to takeaway from our analysis is that the mental models that the many of the developers shared came from real-world applications that they engage with, sometimes on a daily basis. We argue that the idea of introducing usable abstractions at the programming level, the level at which the developers are coding, is far too late. Not only do developers not know how to use cryptographic APIs, they don't even know what they need the cryptographic libraries for. The purpose of newer cryptographic APIs designed to high-abstractions is to improve usability by essentially being an out-of-the-box solution that shields the everyday developer from the low-level cryptographic primitive side of secure programming. These are solutions built on the understanding that developers do not want to engage with low-level implementations of cryptography. However, if we look at the software development lifecycle, or even the secure development lifecycle, developers actively engage with the design process to understand how the components of the application work together before they start programming. Green & Smith's first principle, considered their golden rule, is to *integrate crypto functionality into APIs*. We argue that this integration co9mes too late, instead we propose that this integration be made at the secure design stage of the life cycle, later discussed in Chapter 5.

4.5 Conclusion

We recruited 20 participants based on the minimum requirement of them having some fundamental understanding of cryptography and the ability to read and write code in C. We interviewed these participants during which they were tasked with designing a secure system that achieved the conditions of *confidentiality*, *authentication*, and *integrity*, conditions achieved by Public Key Cryptography. We observed the participant's design process by implementing the think aloud protocol, asking them to explain their thought process as they go along, and the use of diagramming to help visualise their mental models. Based on the interview transcripts and diagrams, we performed the straussian grounded theory analysis [155]. Through the analysis we identified 4 categories and discovered how they relate to each other. Based on the relationship between the 4 categories, we discovered the grounded theory: *Using Security Goals To Test, Update, and Improve the Developers' Mental Repository of Threats and*

Proposed Solutions.

We found that many of the mental models proposed by the participants were based off their day-to-day experience with application security, such as Participant 14's reference to WhatsApps E2EE (Figure 4.5, Figure A) or Participant 13's making the use of emails to authenticate a user of their system. For those participants who did correctly suggest concepts such as *Encryption & Decryption* and *Hashing*, some of them showed a clear understanding of the inverse nature of keypairs and knew that hashes needed to be compared to assess the integrity of a message. However, a handful of participants suggested the concepts of *Encryption & Decryption* and *Hashing* correctly but had misconceptions when it came to applying them to the task. We concluded these mental models could be used to design abstractions for secure programming, however, because these mental models significantly deviate from the concept of Public Key Cryptography and the API designer's mental models, we cannot say that the abstractions designed would be usable.

Our grounded theory analysis captured the participants thought process while they were designing their secure protocols. This analysis showed how the factors of the *Attacker Mindset*, *The Inverse Nature of Encryption & Decryption* and *Concepts Based on Authentication* helped refine the *Developers' Proposed Solutions*. Our grounded theory challenges the idea that the pursuit of designing usable abstractions for secure programming in the form of cryptographic APIs is enough to help developers with secure programming and that out-of-the-box, high abstraction APIs is the answer. Our analysis shows that developers are capable of correctly identifying relevant threats at a high-level and suggesting relevant solutions as well. The key parallel to draw here is between the grounded theory and the *secure design stage* of the Secure Software Development Lifecycle. The objective should be to introduce the need for cryptographic libraries, not at the *development stage* but rather at the *secure design stage* of the SSDLC. We should begin by defining initial steps for distilling the features needed for this approach.

CHAPTER 5

CRYPTOBRIDGE: A NEED FOR EFFECTIVE CHALLENGES & RESPONSES DURING SECURE SOFTWARE DEVELOPMENT

Developers find cryptographic APIs challenging to use. The usable security research community has tackled the challenge in three different ways: the first being guidance, often in the form of recommendations, to help improve the design of security APIs [67]; the second being cryptographic APIs designed to high-abstractions with the aim of being misuse-resistant [37]; the third being security tools that help developers to write and analyse secure code and use cryptographic libraries safely [91]. These methods address the challenge at the programming level of the software development lifecycle, the stage at which the developers are writing the code. Based on our analysis in Chapter 4, we argue that a step needs to be taken before the developer uses a cryptographic API. We argue that the developers' ability to program securely will benefit from gaining some context as to why they need the cryptographic library in the first place.

Based on our analysis in Chapter 4, we established that concept of usability cannot be achieved through a programming abstraction alone. What is needed is a change in perspective where usability should be viewed as a process rather than an abstraction. The grounded theory from Chapter 4 captured the cyclical process of proposing solutions only to break them with potential threats which lead to the refinement of those solutions only to be broken again through the *Attacker Mindset* until the proposed solution achieved the security goals or the developer had reached the boundary without a viable solution. The task itself showed that developers are capable of engaging with security concepts at the application design stage. Not only this but the mental models elicited during the task highlighted blindspots that can be addressed at the application design stage. So now we should ask how can we apply our grounded theory to the secure software development lifecycle?

We present a conceptual *process view* in CryptoBridge. The purpose of CryptoBridge is to provide context at the application design stage of the secure software development lifecycle, so that the developer understands the need for a cryptographic library. We apply aspects of our grounded theory: *Using Security Goals to Test, Update and Improve the Developers' Mental Repository of Threats and Proposed Solutions* that could be used to extract security goals from the developer's application design architecture. A defined set of security goals and potential threats provide context, similar to how the conditions of *confidentiality*, *authentication*, and *integrity* were framed in Chapter 4, providing the developer with a clear agenda with which they can engage with a cryptographic API to solve. To clarify this further, CryptoBridge serves to integrate our grounded theory with the existing SSDLC and introduces the need for cryptographic libraries earlier in the SSDLC at the application design stage.

Applying our grounded theory, CryptoBridge proposes a set of high-level solutions that are associated with the potential threats identified, similar to the developers mental repository of threats and their associated mental repository of solutions.

Similar to how the developers proposed solutions went through the process of refinement where the developers' provided more and more detail about their solutions, CryptoBridge would need to allow for the ability to detail solutions as well. What is required is a supporting mechanism that gives developers the ability to 'drill-down' into a solution to see which cryptographic libraries could support the solution and how is the solution presented through the cryptographic API. The CryptoMap, that offers this functionality, would be designed with three levels of abstraction. The top level of the map is comprised of common cryptographic tasks in plain english, that correspond to the proposed solutions. The middle level hosts a list of seven cryptographic libraries, the ones studied in Chapter 3, and shows whether or not the proposed solution is supported by these libraries. Finally, the low level is comprised of a series of tables that store the cryptographic primitives and protocols supported by the cryptographic libraries. The low level relates to the middle and top level in terms of relevance and requirement.

5.1 Adding Security to the Software Development Lifecycle

When does a Software Development Lifecycle (SDLC) become a Secure Software Development Lifecycle (SSDLC)?

Back in the 70's, to perform an attack, one would need physical access to a terminal on a machine running the application [157]. During this time, systems were a lot less interconnected; reducing the risk of application security being impacted. As new software development methodologies, such as Agile development, were put into practice over the years, security became an afterthought within the SDLC [157]. One reason for this is because when designing an application the SDLC considers the optimal use of a limited set of resources, such as time, finance, and developers. The developers focus on delivering the most value to customers as early as possible, leaving security requirements for later consideration [43, 137].

At first, these applications were tested after their release only and from then on a yearly basis, giving attackers plenty of time to find and exploit vulnerabilities before they are noticed and patched. In response, companies introduced pre-release security testing as well. This step extended the SDLC by several weeks. Furthermore, the results of the security testing were almost impossible to plan for as the test may report only a few vulnerabilities or hundreds. Resolving for the vulnerabilities could mean significant changes to the code and possibly rethinking the components of the architecture entirely, only for the application to undergo the security test again. Not only can this approach set developers back by weeks or even months but the cost of fixing the security issues could be very expensive.

The SDLC becomes the SSDLC when developers become more cognisant of security concerns at each stage of the SDLC. The SSDLC speaks to a mindset that focuses on the secure delivery of an application and raising security concerns as early as the requirements gathering stage of the lifecycle. The Secure Software Development Lifecycle is comprised of 5 stages:

1. *Gathering Requirements:* Along with gather functional requirements from the various stakeholders, developers also identify security considerations as well.
2. *Application Design:* At this stage the requirements are translated into a plan for what the actual application will look like. Based on the functional requirements and security considerations, a secure architecture for the application is designed.
3. *Development:* While implementing the design, developers focus on making sure the code is well-written and secure by running code reviews and adhering by security guidelines. The code reviews are either performed manually or through other methods such as *static application security testing*.
4. *Security Testing & Code Review:* The application is tested for security while ensuring that the requirements are met.
5. *Deployment:* Security Assessment and Security Configuration. Once the application has been released, it becomes the role of the application maintainers to ensure that any vulnerabilities that come to light be addressed immediately. These vulnerabilities could be a result of post-release security assessments, vulnerable code in in open-source projects used by their application, or tests run by ethical hackers during bug bounty programs.

The change from a SDLC to a SSDLC requires *shift-left testing and applications*. Shift-left testing and applications aim to integrate security into the SDLC as early as possible, helping companies release software on a regular basis while keeping common bugs and security issues at a minimum, ensuring the delivery of secure software. However, a recent report by the Enteprise Strategy Group found that while developers are being encouraged to adopt a more security driven mindset towards their software development lifecycle, they simply do not have the knowledge or the right set of security tools to do so [71].

5.2 SSDLC and the Challenge of Secure Programming

When it comes to the challenge of secure programming, the usable security research community has taken three primary approaches. The first being that of guidance, often in the form of recommendations, for designing security APIs [67]. In Chapter 2, we reviewed the scope of the literature surrounding this type of guidance and documented the impact they made on the design of cryptographic APIs such as CryptLib [128], NaCl [37], The second approach is seen through the development of cryptographic APIs designed to a high-abstraction with the aim of being more usable while ensuring a solid level of security. These high-level cryptographic APIs are designed in response to the poor usability reported against libraries that have low-level cryptographic APIs such as OpenSSL. In these low-level APIs, the developer has to use functions that require a strong level of expertise in cryptography, the developer also has to be knowledgeable enough to make decisions such as what types of algorithms and protocols to use as parameters for their application. However, thinking back to Chapter 3, we found that even high-level cryptographic APIs can be difficult to use and could potentially lead to cryptographic misuse. To help developers use cryptographic APIs securely and prevent cryptographic misuse, the research community presents a third approach in security tools commonly seen in the form of static analysis tools and code generation [10, 62, 91].

The SSDLC is an integral part of the *shift-left* initiative and by extension so are the approaches taken towards addressing the challenge of security programming. The three approaches: *Recommendations for Designing Security APIs*, *High-level Cryptographic APIs*, and *Security Tools* fall within the *Development* and *Security Testing & Code Review* stages of the SSDLC. Although these approaches are a step in the right direction, they raise some issues. The usable security research community holds the idea that developers would benefit from out-of-the-box solutions. This can be seen in the design of high-level cryptographic APIs, where high-level functions encapsulate many low-level cryptographic information [37, 52]. The point of this being that the developer can only use the high-level functions and so the chance of misusing the cryptographic API is much lower. The same point can be made about security tools, such as Cognicrypt, who present a wizard to help walk the developer through the process of selecting from a set of common cryptographic tasks and outputting a block of code that can integrate with their existing code workspace [91].

This out-of-the-box mentality plays into the idea of security considerations being an afterthought in the greater software development lifecycle. The shift-left initiative comes too late. Let's say, for example, that a development group are following a regular SDLC. The first two stages are the same as a SSDLC, *Gathering Requirements* and *Application Design*. If the shift-left initiative begins at the *Development* stage of the SDLC, and the developers are introduced to high-level cryptographic APIs and Security Tools, then the idea that security considerations being an afterthought is enforced. Not only do security considerations become an afterthought but the need for a cryptographic API also becomes an afterthought, leading to challenges with secure programming and in turn cryptographic misuse. Even if security tools were presented at this stage to help developers with secure programming, they will be of little help. The introduction of security considerations, cryptographic APIs and security tools so late into the cycle will only be seen as obstacles slowing down the development cycle.

We have established that SSDLC is a shift-left initiative that works towards integrating security at every stage of the SDLC. With this change security considerations are made during the *Requirements*

Gathering stage and the *Application Design* stage. However, we argue that unless the use of cryptographic APIs and security tools is introduced earlier in the SSDLC at the *Application Development* stage, the developers will still face the same challenges with secure programming as they would face during a regular SDLC.

5.3 Applicable Insights from Chapter 4

The task in Chapter 4 was a protocol design task framed as a task of designing a secure application. The participants were asked to design a secure application that could help Alice and Bob send messages securely to each other while achieving *confidentiality*, *authentication*, and *integrity*, the conditions of Public Key Cryptography.

The grounded theory: *Using Security Goals To Test, Update and Improve the Developers' Mental Repository or Threats and Proposed Solutions* was indicative of how developers approach the *Application Design* stage of the SSDLC. In contrast to the out-of-the-box approach seen in the design of high-level cryptographic APIs and security tools, the grounded theory showed developers engaging in the *Application Design* stage of the SSDLC. The tasks also showed that methods such as thinking aloud and diagramming helped developers clearly articulate and define a set of potential threats and related solutions, which in turn resulted in their proposed systems.

The *Attacker Mindset*, one of the four categories that make up the grounded theory, is something that comes naturally to almost all the participants. We can draw a parallel between the elicited *Attacker Mindset* from our grounded theory analysis and the use of threat modeling during the *Application Design* stage of the SSDLC.

Although threat modeling resembles the *Attacker Mindset* elicited from our grounded theory, it does not bridge the gap between the design of the application and the cryptographic library that will be used to implement that design. During the *Application Design* stage, the developers can come up with a set of potential threats to consider but this only partially helps them when using a cryptographic library. To select a cryptographic library, the developer needs to know if the library offers solutions to help mitigate the potential threats they have identified. What form does this solution take, is it written as 'symmetric encryption', 'asymmetric encryption', 'hashing' or something else. They need to translate these terms to fully understand if the function supports their identified threats or not. An example of this can be seen in the documentation for Libsodium, where the documentation is categorised under cryptography concepts as oppose to security tasks that developers may need to perform [52]. Once the cryptographic library is selected, the developers need to actually use the functions provided by the cryptographic library. To use these functions the developer needs to be familiar with the parameters required and in what context the functions should be used. These parameters are derived from the cryptographic concept upon which these function are based. Documentation helps and sometimes cryptographic libraries offer default secure algorithms for the parameters, but we argue that this is a half-measure. Developers may need to adhere to the security guidance provided by their organisation, they may be required to make choices as to which protocols to use. A good example of needing to know strong algorithms from weak can be seen when developers are tasked with defining ciphersuites for their application.

To address these challenges we propose introducing the use of cryptographic libraries earlier in the Secure Software Development Lifecycle from the *Development* stage to the *Application Design* stage.

5.4 CryptoBridge in a Nutshell

We present CryptoBridge, a shift-left initiative that aims to introduce the use of cryptographic libraries in the *Application Design* stage of the Secure Software Development Lifecycle. CryptoBridge is a series of steps that incorporates the aspects of the *Application Design* and *Development* stages together with the aim of providing developers with some context as to how a cryptographic library can help secure their application. We explain the process in seven stages below and to help better understand the process, we will use the example of a simple application:

Stage 1: Designing the Application Architecture

The process begins with the developers using an interactive diagramming interface to visualise the architecture of their application. This interface is based on the benefits seen from the use of diagramming in the protocol design task in Chapter 4. The purpose of this interface is to present a visual representation of the application architecture, allowing developers to fully comprehend the structure of the system and how each component interacts with the others. This interface serves as the highest level of abstraction of CryptoBridge. The developers can select from a range of nodes, such as databases, web services, mobile applications, and draw connections between them showing how they relate to one another. It's important to remember that at this stage of the SSDLC, the developers are in the process of translating the functional requirements into an actionable plan in the form of an application design. The interface will closely resemble the structure of a *Data Flow Model* (Figure 5.1).

Stage 2: Define Data Types

CryptoBridge provides a 'drill-down feature', so that for each node of the architecture, the developers can define the data types that will be stored there. For example, the 'Accounts DB' below stores the credentials and other personal information about the users of the application once they have registered. The personal information can include names, emails, passwords used for the application (Figure 5.2).

Stage 3: Identify Potential Threats

The stages so far would be the same, regardless of whether the developers were using a SDLC or a SSDLC. It is at this stage where the SDLC and SSDLC begin to differ. After seeing developers use a mental repository of potential threats to design a system for the conditions of PKC in Chapter 4, we propose a feature that analyses the application architecture and identifies potential threats based on

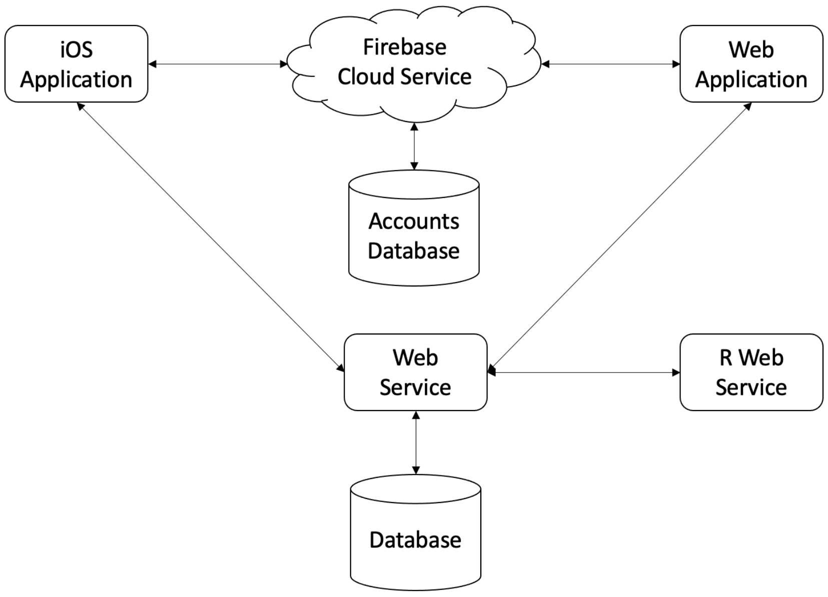


Figure 5.1: An Example of an Application Architecture to Explain the CryptoBridge Process

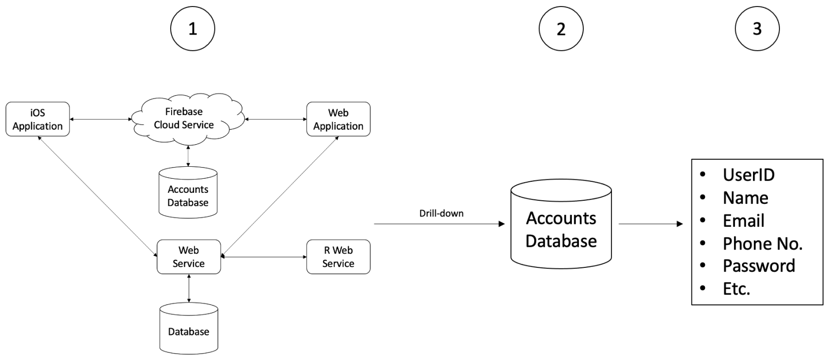


Figure 5.2: Defining Data Types Held Within the Accounts Database

a repository of threats to look for. This process would work in a similar way to how TS4J is used in Cognicrypt [91]. The static analysis functionality provided by Cognicrypt analyses the code against rules defined using TS4J, rules defined by cryptographic experts. TS4J is a fluent interface for defining computing typestate analysis [18]. Only the difference between TS4J and our threat analysis feature being the stage of the SSDLC at which they are implemented. This feature notifies the developers of vulnerabilities in their application design such as unsecure connections and unhashed passwords (Figure 5.3).

Stage 4: Define Security Goals

In our grounded theory from Chapter 4, the participants also had a mental repository of solutions associated with their repository of threats. We reflect this relationship in the steps of CryptoBridge. For each threat in our repository, there are associated solutions offered. For example, for an unsecure

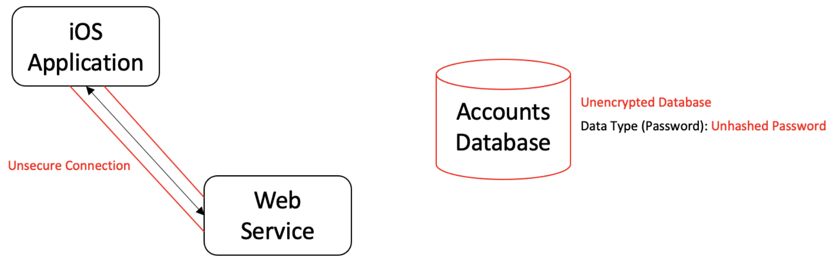


Figure 5.3: An Example of Potential Threats Identified Through the CryptoBridge Process

connection, the associated solution of establishing a secure connection would be suggested (Figure 5.4).

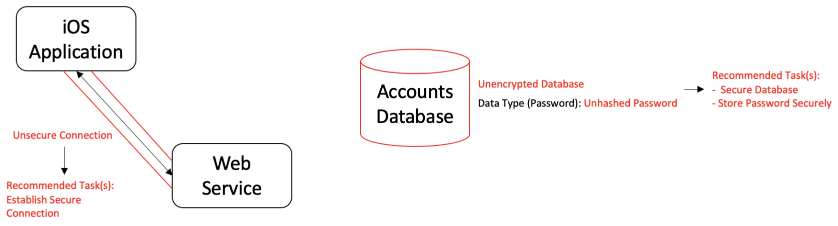


Figure 5.4: An Example of Potential Threats Identified and Associated Solutions Proposed

Stage 5: Using the CryptoMap for Assessment & Selection of Cryptographic Libraries

An argument can be made that there are existing applications that can achieve the features discussed in the stages so far. However, from this stage onwards, we begin integrating the use of cryptographic libraries with the *Application Design* stage.

In the supporting mechanism, CryptoMap, we see a mapping of the proposed solutions to the functionality supported by cryptographic APIs. The purpose of CryptoMap is to present the developers with a choice of cryptographic libraries based on whether or not they support the suggested solution. Not only are the developers presented with a choice but they are also given recommendations. For example, a cryptographic library like OpenSSL supports 17 hashing algorithms. Without some expertise in cryptography, how does the 'everyday' developer tell a strong algorithm from a weak one [76]. This is where CryptoMap could make recommendations related to the suggested solution for the identified threats. We will explain the inner workings of the CryptoMap aspect of the CryptoBridge process in more detail later.

Stage 6: Securing the Application Architecture

In 2017, Poppele et al. [131] presented a classification of over 700 cryptographic libraries chosen based on how current and popular they were. The purpose of the classification was to help both experienced and inexperienced developers select a cryptographic library. Recently, in 2022, we reviewed this raw data for accuracy and made updates if any libraries did so too. We then mapped the data. The CryptoMap allows developers to ‘drill-down’ into three levels. The low level represents the data initially collected by Poppele et al. [131], later updated by us. This level holds cryptographic primitive data such as the types of algorithms and protocols supported by the libraries. The middle level is the cryptographic library identification level. At this level, the CryptoMap holds information about 753 libraries collected from 17 languages. This information includes general points such as what type of cryptographic library it is, where it could be a *standalone*, *wrapper*, *fork*, or a *reimplementation*. Other information includes: Lines of Code (1000s), Contributors, Links etc. The top level holds a series of common security tasks, such as *Establish a Secure Connection*, *Store Password Securely* and a few other. The tasks in the top level could be derived from common tasks used in other security tools such as Cognicrypt [91] (Figure 5.5). These top-level tasks would be associated with the security goals defined in Stage 4 of the CryptoBridge Process

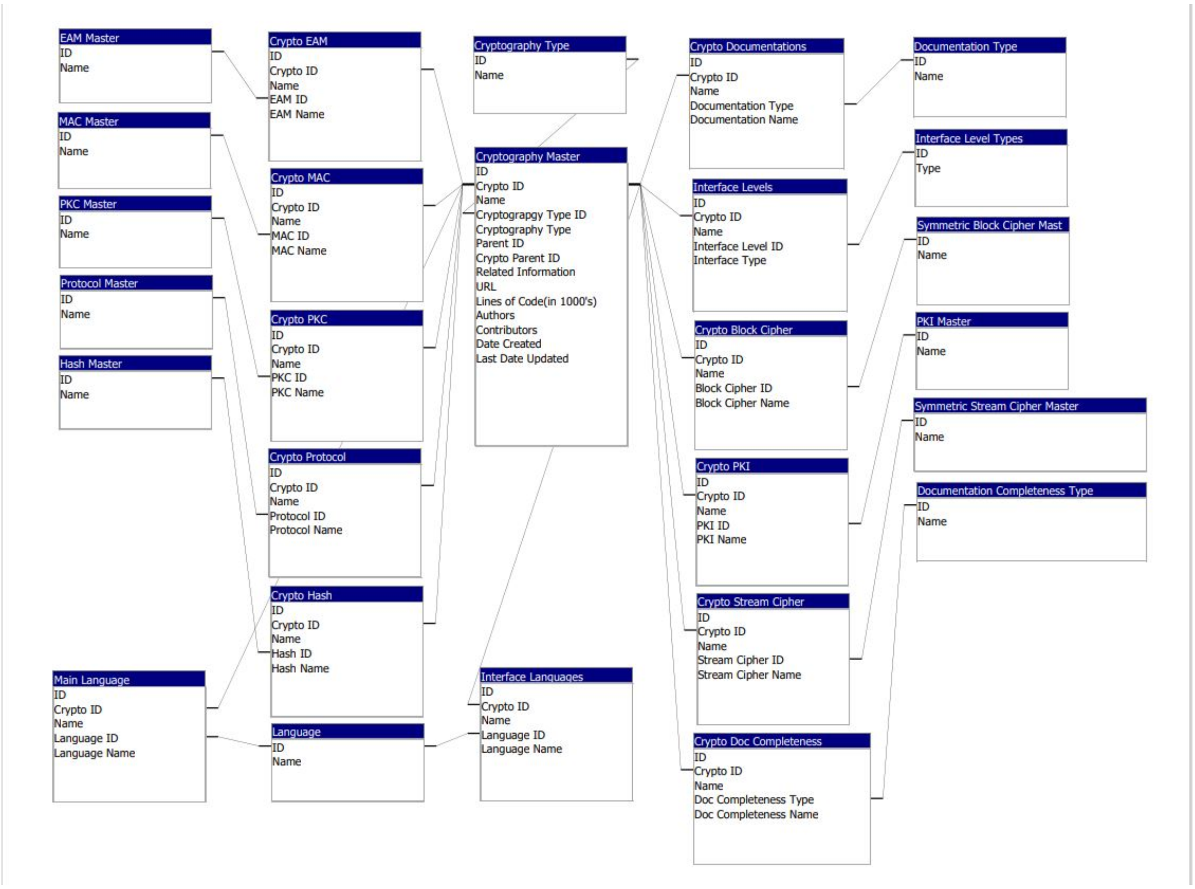


Figure 5.5: CryptoMap

CryptoMap would allow the developers to select from a range of algorithms that have been categorised under tables such as *Crypto Stream Cipher*, *Crypto Block Cipher*, *Crypto PKI* and many others. CryptoMap could also offer information on *Documentation Completeness* which can help inform the ‘everyday’ developer before they start using the cryptographic library.

By this stage CryptoBridge has completed its task by showing the developers how their application looks, the potential threats identified at each component of their application, a suggestion of solutions associated with the potential threats identified, and recommendations in terms of cryptographic library support and secure algorithms and protocols to choose from when using the library. At this stage, the developers are given to choice to either accept the security recommendations or to go through the process of using the map and select from the cryptographic algorithms and protocols offered (Figure 5.6).

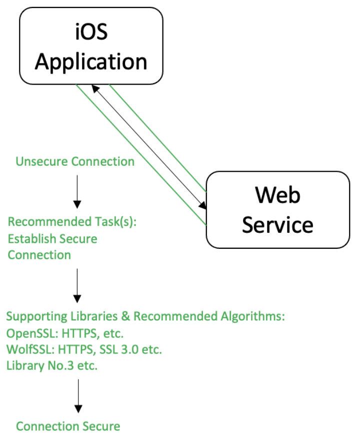

iOS Application
Web Service
Unsecure Connection
Recommended Task(s): Establish Secure Connection
Supporting Libraries & Recommended Algorithms:
OpenSSL: HTTPS, etc.
WolfSSL: HTTPS, SSL 3.0 etc.
Library No.3 etc.
Connection Secure

Figure 5.6: An Example of Resolving a Potential Threat During the Application Design Stage

Stage 7: Using the Cryptographic API

By this final stage, all security considerations are made along with a complete integration of a cryptographic library with the *Application Design* stage of the SSDLC. The developers have a security plan in place and can move onto the *Development* stage of the SSDLC with a clear plan.

5.5 Conclusion

We introduced this chapter with the conclusion that an abstraction on its own is not enough to ensure usability. Instead, we needed to view usability as a process as we saw in the grounded theory explained in Chapter 4. The grounded theory highlighted an important relationship between the developers' perception of threats and associated solutions. The Secure Software Development Lifecycle is a shift-left initiative in itself, with the aim of integrating security at every stage of the software development lifecycle. We proposed CryptoBridge, a shift-left initiative within the Secure Software Development Lifecycle with the aim of introducing the need for cryptographic libraries through a series of steps, a process to be followed during the *Application Design* stage of the SSDLC.

Chapter 6

Conclusion

The central research hypothesis of this thesis was that many instances of cryptographic misuse are due to misalignments between the developer's mental model and the functionality presented through cryptographic APIs based on the API designer's mental model of standard cryptographic processes like Public Key Cryptography. After being tasked with designing a security protocol that should aim to achieve the conditions of Public Key Cryptography, we elicited 11 mental models from developers. We concluded that although these mental models could be used to design abstractions for secure programming, they would not result in usable abstractions. The same study, in Chapter 4, also resulted in a grounded theory that captured the developer's thought process. This grounded theory served as a key finding as it changed our perspective of usability, as a concept. Instead of trying to achieve usability through abstraction alone, we found that usability is a process that we applied in Chapter 5 to the SSDLC through a shift-left initiative in CryptoBridge. Before we took the mental model approach, however, we performed a systematic literature review on the development and validation of guidance surrounding the design of security APIs. We carried out our own empirical validation study on a set of 10 principles proposed by Green & Smith, where we identified 16 usability issues faced by developers while they used 7 cryptographic libraries and mapped these usability issues against the 10 principles of Green & Smith which resulted in 4 novel usability smells. In this chapter, we revisit the overarching objectives defined in the introduction of this thesis and discuss the findings, key insights, and future works.

6.1 Thesis Objectives Revisited

6.1.1 What guidance is available for designing security APIs and what is its empirical foundation?

Open Research Questions:

- What do current recommendations focus on and what types of challenges do they address?
- How, and to what extent, have various recommendations been validated?

We found that both *API Designer* and *Security API Designer* papers offered recommendations that primarily focused on *Construction*, the technical features of software and APIs, with a focus on structuring code. The *Construction* category was addressed by 57% of the API designer recommendations and 36% of the Security API designer recommendations. The *API Designer* and *Security API Designer* papers also focused on providing guidance for how software and APIs come to be understood and practiced by people, with 21% and 24% of their recommendations falling under the category of *Understanding*, respectively. We found these areas of focus reasonable because API-related recommendations would be likely to deal with the interface with code. *Software engineering* literature placed a greater emphasis on *Understanding* with 54% of its recommendations than *Construction* with 25%. *Security engineering* guidance placed its focus into *Assessment* (The quality and testing of software and APIs - 25%) and *Requirements* (The development of requirements for software and APIs - 18%). It was interesting to find that the recommendations that focused on *Organisational Factors* (How organisations respond to developing software and APIs and interface with external factors) came almost exclusively from *Security engineering* papers. These recommendations encouraged steps such as *Risk Assessment*, *Developer Training* and *Incident Response*. Unsurprisingly, much of this guidance came from corporate literature such as Microsoft [102], BSIMM [25], and OWASP documentation [121].

To understand how, and to what extent, had recommendations been validated, we assessed the ancestries of recommendations and identified 5 different ways by how they came to be. These were shown through our ancestor-descendant relationships: *Distillation*, *Borrowed*, *Adaptation*, *Comparison*, and *Empirical Validation*. We found that overall, 52% of the papers were a part of an *Adaptation* relationship, whereas only *22%* of the papers engaged in *Empirical Validation*. Although 22% is not terribly low, we do encourage empirical validation of the literature found to help prevent the propagation of ineffective guidance. Of the 13 *Security API Designer* papers found, only 3 of the papers were empirically validated.

6.1.2 What do software developers struggle with when using cryptographic APIs?

Open Research Questions:

- What types of challenges do developers face while they use cryptographic APIs?
- To what extent do principles, aimed at improving the usability of cryptographic APIs, address the challenges faced by developers?
- To what extent does the design of cryptographic APIs conform to such principles?

We performed a thematic analysis of 2,491 Stack Overflow questions and responses posted by developers who were using 7 cryptographic libraries. As a result of this thematic analysis, we identified

16 usability issues with which developers struggle, which we further categorised into 7 themes. We found *Missing Documentation* to be a common issue. It shows the challenges developers' face in the initial stages of using a cryptographic library, as they are unable to find the necessary documentation to support them in using the cryptographic API. For example, SJCL (a JS cryptographic library) only offers code as a resource in their API documentation. So this usability issue could be easily addressed by adding documentation to support the use of the cryptographic API. The use of a forum such as Stack Overflow can be explained through issues that fall under *How should I use this?*. There are many instances under which developers have a specific feature in mind for an application and want to know how to securely implement this feature. The count of Stack Overflow questions is high for OpenSSL because the developers believe that other developers have already implemented the feature and have made their code available. So the developers resort to finding the specific implementation on forums such as Stack Overflow instead of using the documentation provided by the cryptographic library, if provided at all. The idea that the security task the developer had in mind has been coded by others and made available gives a false sense of security that the code must work. This false sense of security could be the reason why there are many questions recorded where developers show examples of broken code and ask for guidance with debugging. Its important to note that the responses usually come in the form of task-based code examples showing how to correctly implement the feature and this is something developers respond to very well. This is a key point to consider because the developers naturally create security goals in the form of tasks which relates to aspects of our grounded theory in Chapter 4 in turn informing our process view defined in Chapter 5.

Having identified the 16 usability issues, we map them against Green & Smith's 10 principles (Figure 3.1), and as a result identify 4 usability smells that are indicative of one or more of the principles not being fully observed by the cryptographic libraries. The purpose of the usability smells is to help improve the usability of a cryptographic library based on where the developer appears to struggle. We found that issues that fall under the usability smells: *Needs a super sleuth*, *Confusion reigns*, and *Needs a post-mortem* are addressed by Green & Smith's principles. However, our fourth usability smell; *Doesn't play well with others* is comprised of issues such as: Build, Compatibility, and Performance issues, is not well covered by Green & Smith's principles. We apply our mapping to the cryptographic libraries to see to what extent they show signs of our usability smells. All libraries showed signs of *needing a super sleuth* with OpenSSL and Bouncy Castle users struggling with a lack of documentation. Bouncy Castle showed strong signs of *confusion reigns*, where developers where questioning whether or not it was appropriate to use the library in the first place. These developers also showed issues with the abstractions provided. We found this surprising as as Bouncy Castle integrates with the Java Cryptography Architecture which offers a standard API for cryptographic libraries in general. Furthermore, Bouncy Castle offers its own API and supports multiple languages, which may give a reason as to why developers find the library confusing to use, as they are faced with too much choice. This further strengthens our point that in addition to abstraction, usability should be approached with a *process view*. Developers relate to defining security goals and defining a need for cryptographic functionality as oppose to being presented with too many resources but no agenda.

6.1.3 What are the mental models of the software developers who use cryptographic APIs?

Open Research Questions:

- What are the developers' mental models surrounding integral cryptographic concepts like Public Key Cryptography?
- What are the misconceptions seen in the developers' mental models and how do they deviate from the cryptographic concept?
- When put in the position of Ellis, tasked with designing a security protocol that achieves the conditions of Public Key Cryptography, what are the mental models held by developers and how do these compare to Ellis?

We elicited 11 mental models from developers after they were tasked with designing a system that allowed two parties to securely exchange messages while ensuring that their system achieves the conditions of Public Key Cryptography: *confidentiality*, *authentication*, and *integrity*. We found that many of these mental models were a result of the developers day-to-day experience with application security, such as E2EE seen through WhatsApp, or two-factor authentication. Many of the participants presented an understanding of the inverse nature of encryption & decryption, although not always in the form of keypairs. Cryptography students referred to examples of the Caesar cipher or a mathematical explanation of the RSA algorithm. Mental models such as *access control* were used to address the condition of *confidentiality*, an alternative model to that of *encryption & decryption*. When it came to the condition of *authentication* participants referred to day-to-day applications that request their emails or pins as a form of authentication, and integrated these mental models into their system design. Many of the participants correctly brought up the concept of hashing and the need to compare hashes to assess the integrity of a message, however, when asked to explain how hashing works, some of the participants showed clear misconceptions. For example, there were instances where a participant suggested hashing half the message instead of the whole message, another participant suggested hashing a combination of the message and the sender's public key. The task allowed us to not only pinpoint the misconceptions developers had but to also see how the developers mental model plays a part in the greater task of designing a security protocol.

Open Research Questions:

- What is the developers' thought process when asked to design a security protocol?
- Does the developer show signs of an 'attacker mindset' when designing security protocols?
- If so, what are the threats raised by these developers and how do they mitigate them?
- How does the developers' thought process impact their mental models?

We clearly defined the developers' thought process through our straussian grounded theory methodology. The grounded theory, titled: Using Security Goals To Test, Update and Improve the Developers' Mental Repository of Threats and Proposed Solutions, comprised of four categories that interacted with each other. The categories being the *Developers' Proposed Solutions*, *Attacker Mindset*, *The Inverse Nature of Encryption & Decryption* and *Concepts Based on Authentication*. Not only did the developers show signs of an *attacker mindset* but it served as a means of challenging the *developers' proposed solutions* to see if they break or not and if so, the developers would further detail their proposed solutions and suggest changes to mitigate the potential threats they identified while achieving the conditions of *confidentiality*, *authentication*, and *integrity*.

6.1.4 What are the misalignments between these mental models and correct usage of the API?

Open Research Question:

- How do the developers' mental models compare to low- and high- cryptographic API implementations, such as OpenSSL and Libsodium, of Public Key Cryptography?

In the discussion of Chapter 4, we talk about how OpenSSL offers an a high-level cryptographic API in the Envelope API. The abstraction of the postal system was only raised by Participant 20. Other participants such as Participants 10 and 11 brought up the act of physically signing a letter when addressing the condition of *authentication* but this was the extent of the similarities. An important similarity to note is that of the developer's *encryption & decryption* mental model to the encryption & decryption seen through the crypto_box API of Libsodium. This similarity is not only limited to the mental model but also the category of *The Inverse Nature of Encryption & Decryption* that many developers seemed to understand.

6.1.5 Can the developers' mental models be used to design usable abstractions for secure programming?

Open Research Questions:

- Once the developers' mental models have been elicited, how do we integrate them with the abstractions seen through existing cryptographic APIs?
- Do developers' mental models exhibit counterproductive implications?

At the end of Chapter 4, we concluded that the developers' mental models could be used as abstractions but because these mental models deviated so much from the standard way in which the concept of Public Key Cryptography works, we could not say that these abstractions would be *usable* for secure programming. There were very little similarities between the developers' mental models and existing cryptographic APIs so trying to integrate these abstractions with existing cryptographic APIs would be a futile exercise. A valuable insight provided by the developers' mental model, however, is that were able to pinpoint exactly where misconceptions took place between their mental models and the correct way in which Public Key Cryptography works. The study changed our perception of what usability means. In Chapter 5, we presented a change in perspective from seeing usability from an *abstraction view* to a *process view*. In Chapter 5, we present a shift-left initiative in CryptoBridge, which presents intial steps in the form of a series of steps that take place within the Secure Software Development Lifecycle at the *Application Design* stage. The aim of the shift-left initiative is to introduce the need for cryptographic libraries earlier in the SSDLC from the *Development* stage to the *Application Design* stage. This approach provides developers with some context as to why they need a cryptographic library and allows them to define security goals and a plan of action before using the cryptographic API.

6.2 Future Work

6.2.1 Studying the Impact of Abstraction in Cryptographic API Design

Observing developers as they attempt to perform cryptographic tasks using high- and low- level cryptographic APIs would build on the work already done in Chapter 4. Specifically, the same study group, from the study of Chapter 4, could be asked to implement Public Key Cryptography using the low-level interface of OpenSSL and the high-level interface of Libsodium. This study would show how developers navigate an interface designed based on the API designer's mental model. Would the developer attempt to bridge the gaps between their own mental model and the API designers? Would the developer choose to break their mental model and adopt the one shown through the cryptographic API? Is that even possible for the 'everyday' developer who does not have expertise in cryptography? These are questions that can be answered from a study like this.

6.2.2 Studying the integration of security tools with cryptographic libraries

When referring back to the main approaches taken by the usable security research community for improving the practice of secure programming, we identified three general ways: *recommendations for security APIs*, *high-level cryptographic APIs*, and *security tools*. Chapter 2 is dedicated to studying guidance for designing security APIs through a systematic literature review. Chapter 3 studies the challenges faced by developers while they use 7 cryptographic APIs, ranging between high- and low-abstraction designs. We discuss the advances made in the form of security tools during Chapter 2 and also during Chapter 5. However, there is scope for us to research into the integration of existing security tools, like Cognicrypt [91], and cryptographic libraries. Does the combination improve the functional security of an application? What is the impact of this integration on usability? How seamlessly does this integration work with an existing software development lifecycle? These are questions that can be raised while planning this study.

6.2.3 Tool Support for CryptoBridge

In Chapter 5, we propose a shift-left initiative in CryptoBridge, a first step that offers a series of steps with the aim of introducing the need for cryptographic libraries earlier in the SSDLC, providing developers with the ability to define security goals and a plan of action before using cryptographic APIs. This process requires tool support, which we see being implemented as part of a future work.

6.2.4 User Study with CryptoBridge

To test the effectiveness of the approach, detailed in the process of CryptoBridge, we would need to observe developers across both the *Application Design* stage and the *Development* stage of the Secure Software Development Lifecycle. Futhermore, we argue that it would be more effective if we studied developers groups instead of a developer at a time. This is because the *Application Design* stage is an engaging part of the lifecycle where developers come together to discuss the architecture of the application. This approach could build on the grounded theory from Chapter 4, where one developer played the roles of testing, updating, and improving by themselves. In a developer group we may see these roles get naturally taken up by different members of the development team. Then, we can compare our analysis of their interactions to our existing grounded theory to see if we need to update the process of CryptoBridge to cater to the process of a development group as oppose to one developer. Also, a developer study can also answer whether or not tool support for CryptoBridge is actually necessary or not.

6.3 Concluding Remarks

At the end of this thesis, we find that usability cannot be achieved by abstraction alone. Instead, we should look at usability as a process which begins far earlier than the development stage of the secure software development lifecycle. Through our systematic literature review, the usable security research community is encouraged to engage with and empirically validate a number of works relevant to the design of security APIs. The usability smells can also be used as a means to validate security APIs and improve their usability by making simple changes such as improving the clarity of documentation or providing examples of code. The mental model approach, taken in this thesis, showed the misalignments between the developer and integral concepts like Public Key Cryptography, and explains why developers find it difficult to use cryptographic APIs. Our work provides a foundation upon which mechanisms can be designed to address the misalignments seen through the elicited mental models and further support the usability of cryptographic APIs and help developers write secure code.

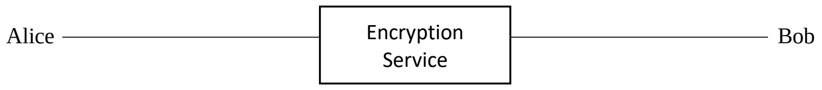
APPENDIX A

APPENDIX A

A.1 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 1

Participant 1

Condition 1: Confidentiality



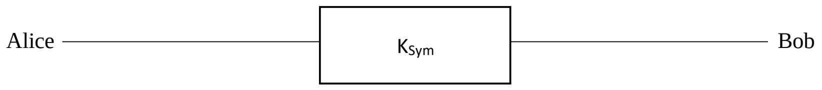
//

Threats:

- Eavesdropping (Confidentiality/Authentication)
- Tampering (Integrity)

Encryption Service:

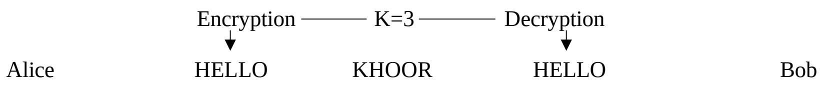
- Symmetric Key Cryptography



//

Additional Security:

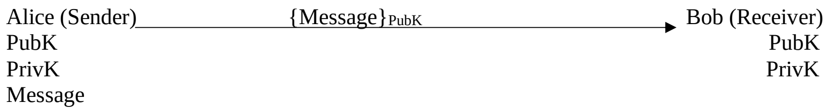
- Substitution Pattern
- Caesar Cipher



//

Threats:

- Gaining Access
- Man-In-The-Middle



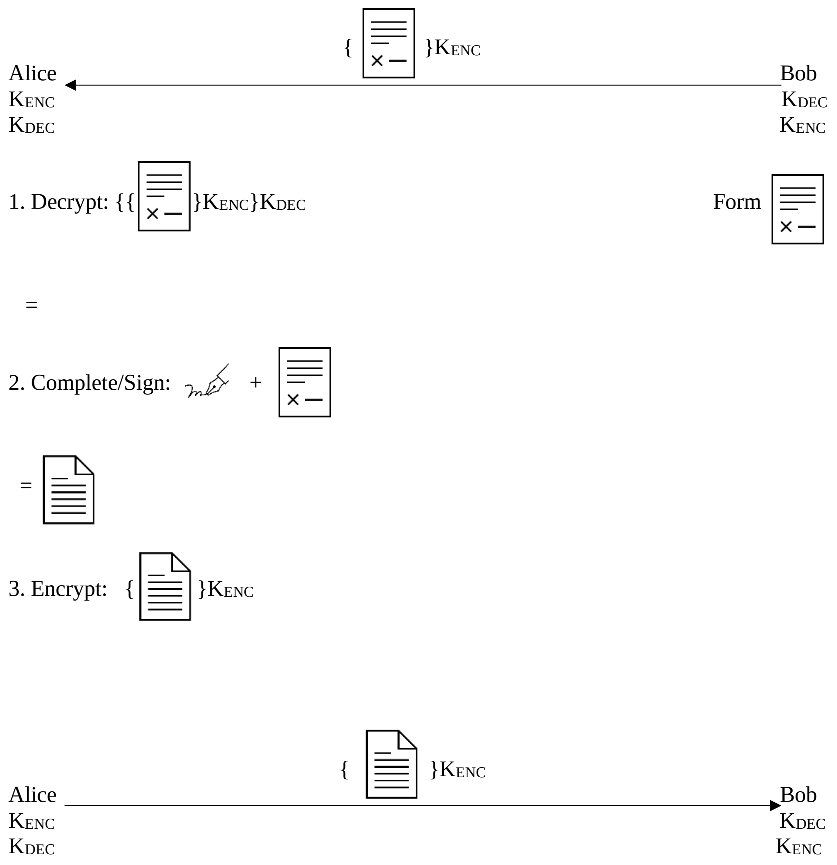
$\{Message\}_{PubK}$
$\{\{Message\}_{PubK}\}_{PrivK}$
= Message

//

Participant 1

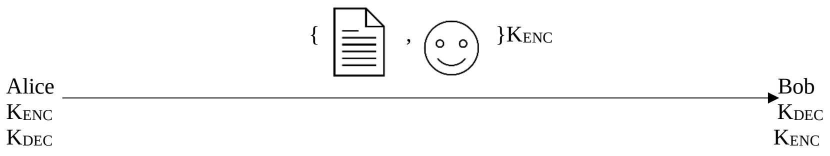
Condition 2: Authentication
Threats:

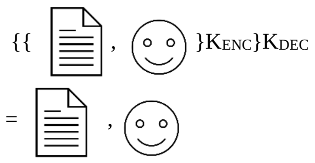
- IP Spoofing
- Imitation

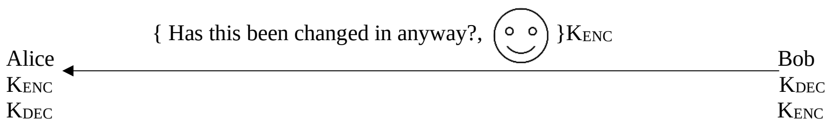


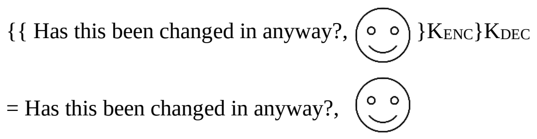
Participant 1

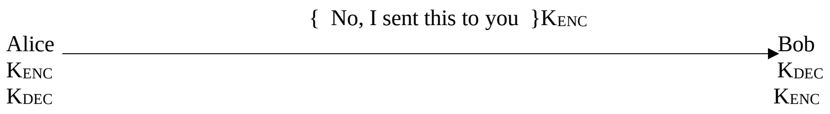
Revised:








{{ Has this been changed in anyway?, }K_{ENC}}K_{DEC}
= Has this been changed in anyway?,

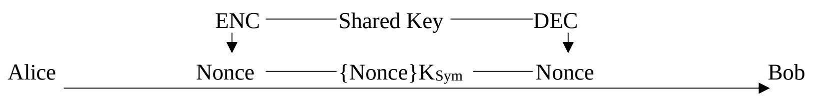


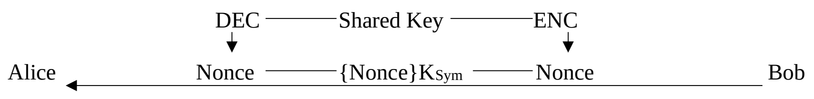
//

Participant 1

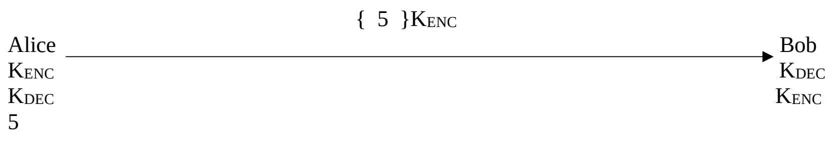
Condition 3: Integrity
Threat:

- Man-In-The-Middle (MITM)
- Playback Attack
- Tampering

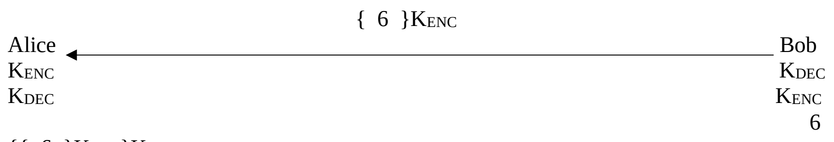




//



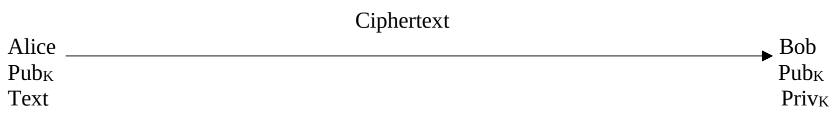
{{ 5 }K_{ENC}}K_{DEC}
= 5



{{ 6 }K_{ENC}}K_{DEC}
= 6

Because 5 != 6, the shared key has been compromised.

//



Participant 1

$\{Text\}Pub_K = Ciphertext$

$\{Ciphertext\}Priv_K = Text$

Eve
Pub_K

//

$$\begin{tikzcd}
\text{Alice} \arrow[r, "\{K\}Bob_{PubK}"] & \text{Bob}
\end{tikzcd}$$

Alice
Bob_{PubK}
Text = K

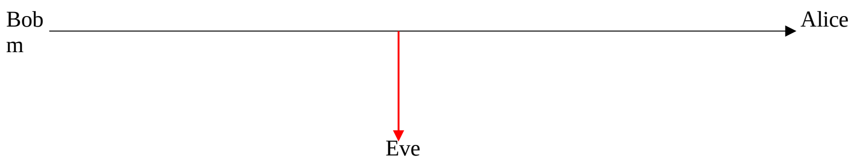
Bob
Bob_{PubK}
Bob_{PrivK}
$\{\{K\}Bob_{PubK}\}Bob_{PrivK} = K$

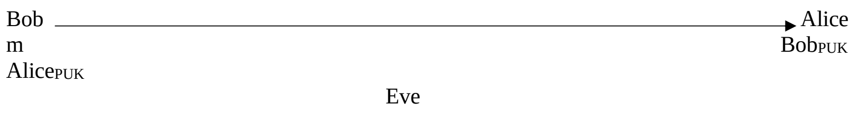
A.2 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 2

Participant 2

Threats:

- Man-In-The-Middle (MITM)





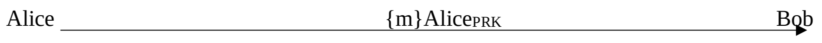
Conditions Addressed:

- Confidentiality ~
- Integrity ~

//

Condition Addressed:

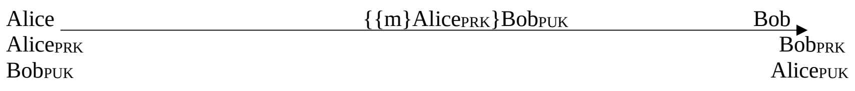
- Authentication



//

Conditions Addressed:

- Confidentiality
- Authentication



$\{\{m\}Alice_{PRK}\}Bob_{PUK}$
$\{\{\{m\}Alice_{PRK}\}Bob_{PUK}\}Bob_{PRK}$
$= \{m\}Alice_{PRK}$

$\{\{m\}Alice_{PRK}\}Alice_{PUK}$
$= m$

Participant 2

Conditions Addressed:

- Integrity

Threats:

- Hash Collision Attack (Side Note: Low Probability)

Message 1 != Message 2

$$\begin{tikzcd}
\text{Message 1} \arrow[r] & \boxed{\text{Hash Function}} \arrow[r] & \text{Hash 1} \\
\text{Message 2} \arrow[r] & \boxed{\text{Hash Function}} \arrow[r] & \text{Hash 2}
\end{tikzcd}$$

Two Parts:

- Actual Message
- Checksum/Hash

$$\begin{tikzcd}
\text{Alice} \arrow[r, "\text{Message, Checksum/Hash}"] & \text{Bob}
\end{tikzcd}$$

Message
Checksum/Hash

```
If (Hash(Message)) == Checksum/Hash{
        "Integrity Intact"
} else {
        "Integrity Compromised"
}
```

A.3 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 3

Participant 3

Initial Thought:

- RSA

Alice	Bob
m	SK2
SK1	PK2
PK1	

$$\begin{tikzcd}\boxed{\text{Asynchronous Mechanism}} \arrow[r] & \text{(RSA Key Distribution)}\end{tikzcd}$$

$$\begin{tikzcd}\text{Alice} \arrow[r, shift left, "PK1"] & \text{Bob} \arrow[l, shift left, "PK2"]\end{tikzcd}$$

//

Alice	Bob
m	SK2
SK1	PK2
PK1	PK1
PK2	

$$\begin{tikzcd}\text{Alice} \arrow[r, "\text{Encryption}\ \{m\}_{PK2}"] & \text{Bob}\end{tikzcd}$$

$\{\{m\}_{PK2}\}_{SK2}$
$= m$

//

Integrity:

- Digital Signature

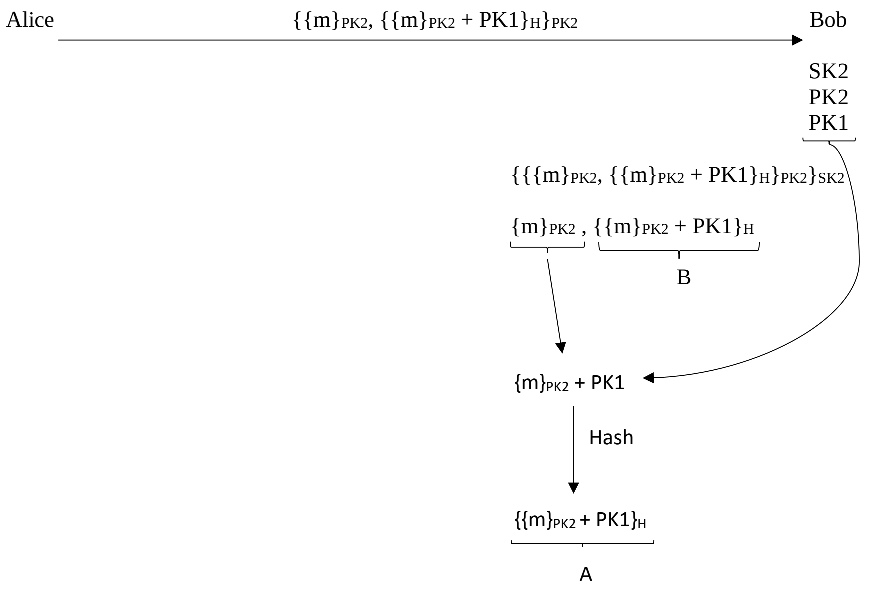
Alice:

1. Encrypt message with PK2
   $\{m\}_{PK2}$
2. Separately, concatenate encrypted message with PK1
   $\{m\}_{PK2}$ + PK1

Participant 3

3. Hash Step 2
   $\{\{m\}_{PK2} + PK1\}_{H}$

4. Encrypt Step 1 and 3 with PK2
   $\{\{m\}_{PK2}, \{\{m\}_{PK2} + PK1\}_{H}\}_{PK2}$



Alice
$\{\{m\}_{PK2}, \{\{m\}_{PK2} + PK1\}_{H}\}_{PK2}$
Bob
SK2
PK2
PK1
$\{\{\{m\}_{PK2}, \{\{m\}_{PK2} + PK1\}_{H}\}_{PK2}\}_{SK2}$
$\{m\}_{PK2}$, $\{\{m\}_{PK2} + PK1\}_{H}$
B
$\{m\}_{PK2} + PK1$
Hash
$\{\{m\}_{PK2} + PK1\}_{H}$
A

```
If (A == B){
      "Verified Digital Signature"
} else {
      "Not Verified"
}
```

A.4 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 4

Participant 4

Confidentiality:

Threats:

- Eavesdropping

$\text{Alice} \xrightarrow{\{\{m\}_{APRK}\}_{BPUK}} \text{Bob}$

A_{PUK}
A_{PRK}
B_{PUK}

$\{\{\{m\}_{APRK}\}_{BPUK}\}_{BPRK}$
$= \{m\}_{APRK}$

//

Authentication:

Threats:

- Imitation

Approach:

- Digital Signature

$\text{Alice} \xrightarrow{\{m\}_{APRK}} \text{Bob}$

$\{\{m\}_{APRK}\}_{APUK}$
$= m$

//

Integrity:

Threats:

- Tampering

$\text{Alice} \xrightarrow{\{m, \{m\}_H\}_{APRK}} \text{Bob}$

$\{\{m, \{m\}_H\}_{APRK}\}_{APUK}$
$= m, \{m\}_H$

Perform Hash of m
$= \{m\}_H$

Compare $\{m\}_H$ and $\{m\}_H$ for Integrity

A.5 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 5

Participant 5

Initial Thought:

- SSL
- Public/Private Keypairs
- Threat: Eavesdropping

$$\begin{tikzcd}
\text{Alice} \arrow[rrr, "\{m\}_{BPUK}"] & & & \text{Bob}
\end{tikzcd}$$

$\{\{m\}_{BPUK}\}_{BPRK}$
$= m$

Condition Addressed:

- Confidentiality

//

Condition: Authentication

Initial Thought:

- Keypairs (Asymmetric Key Concept)

$$\begin{tikzcd}
\text{Alice} \arrow[rrr, "\{\{m\}_{APRK}\}_{BPUK}"] & & & \text{Bob}
\end{tikzcd}$$

$\{\{\{m\}_{APRK}\}_{BPUK}\}_{BPRK}$
$= \{m\}_{APRK}$

$\{\{m\}_{APRK}\}_{APUK}$
$= m$

//

Condition: Integrity

Threat:

- Tampering

Approach:

- Check Integrity

$$\begin{tikzcd}
\text{Alice} \arrow[rrr, "m, \{m\}_{H}"] & & & \text{Bob}
\end{tikzcd}$$

Perform hash on m.
$\{m\}_H$

If($\{m\}_H == \{m\}_H$){
"Integrity Intact"

Participant 5

```
}else{
“Message has been tampered with”
}
```

A.6 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 6

Participant 6

Defines Situation:

- “Whatever Alice sends to Bob can be seen by Eve.”

Alice Bob

Eve

//

Alice. $\xrightarrow{\text{Purple Key (APUK)}}$ Bob
Red Key (APRK) Purple Key (APUK)
Purple Key (APUK)

Eve

Alice has two keys:

- Red Key: Private Key
- Purple Key: Public Key

//

Participant explains inverse nature of Red/Purple (Private/Public) Keys:

$\boxed{\text{Plaintext Message}}$ Red Key (Encryption) = Ciphertext

$\boxed{\text{Ciphertext}}$ Purple Key (Decryption) = Plaintext

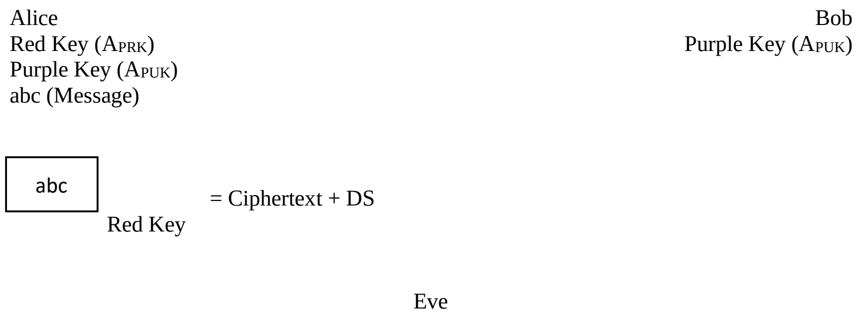
Or

$\boxed{\text{Plaintext Message}}$ Purple Key (Encryption) = Ciphertext

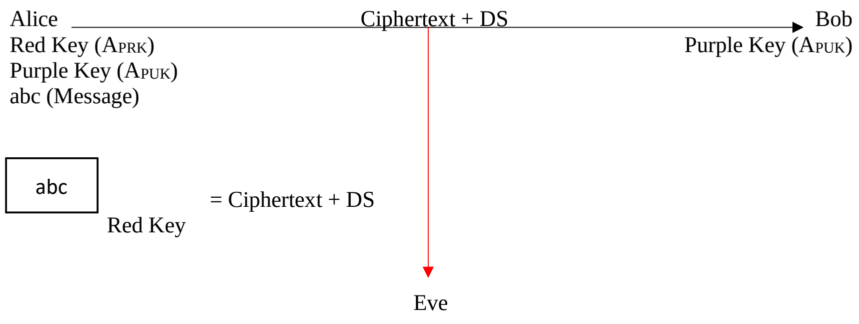
$\boxed{\text{Ciphertext}}$ Red Key (Decryption) = Plaintext

//

Participant 6



1. Alice encrypts plaintext message ‘abc’ with Red Key (APRK) and gets the ciphertext.
2. Alice attaches her Digital Signature, DS, to the ciphertext.
3. Alice sends the combination of the ciphertext and the Digital Signature over the network to Bob. This package is also available to Eve.



//

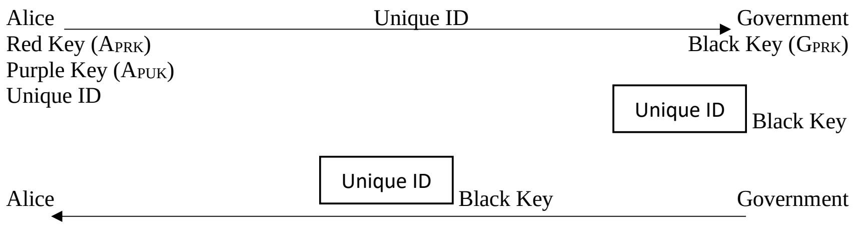
Participant explains Digital Signatures:

- The Digital Signature is created by using Alice’s Red Key (APRK)
- Previous Statement Revised
- The Digital Signature is created by a third-party
- The standard term for this third-party is Certificate Authority (CA)
- Participant chooses not to use the standard term (CA) but instead uses the term ‘Government’

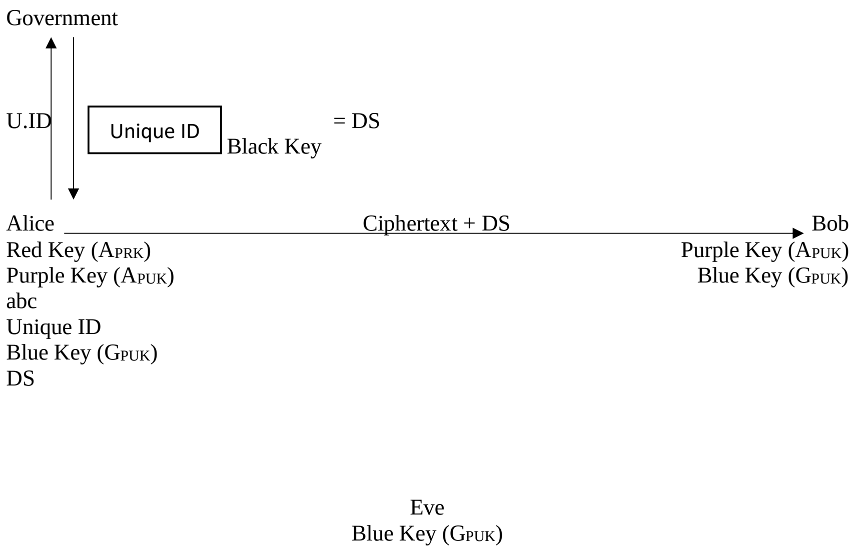
1. Alice sends her Red Key (APRK) (Statement Revised). Alice has her own unique identity which consists of her name, job, etc.
2. Alice sends her unique identity to the Government
3. The Government signs Alice’s unique identity with the Government’s Black Key (GPRK). The Government encrypts Alice’s ID with it’s Black Key (GPRK)
4. The Government sends the signed ID to Alice

Participant 6

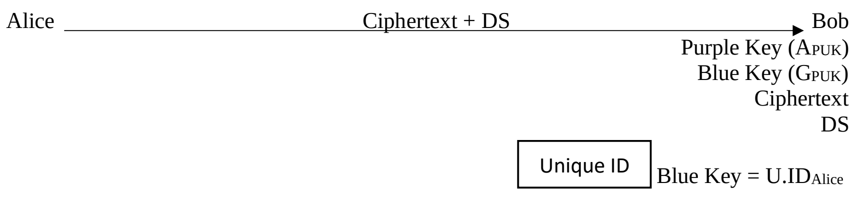
5. Alice uses the signed ID as her Digital Signature



6. The Government has a Blue Key (GPUK), it's Public Key. The Government shares it's Blue Key with everyone. Bob, Alice, and Eve, all have the Blue Key now.



7. Bob verifies Alice's Digital Signature (Condition Addressed: Authentication)
   i. Bob knows that the DS is certified by the Government
   ii. Bob has the Government's Blue Key (GPUK)
   iii. Bob decrypts Alice's DS to access Alice's ID



Participant 6

//

Challenge:

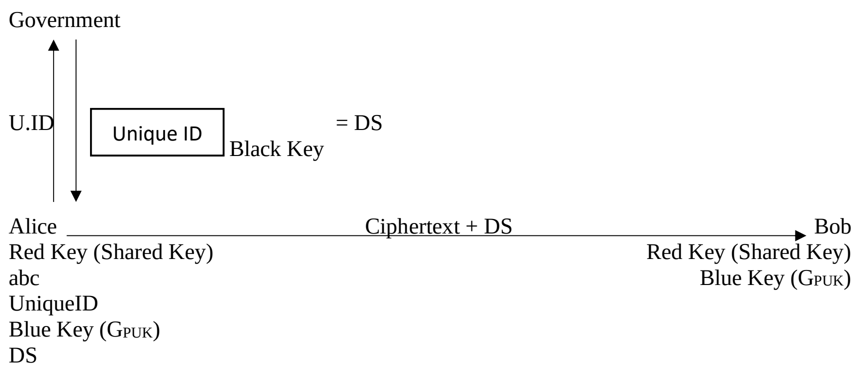
- Does Eve not have Alice's Purple Key (APUK)? (Condition Addressed: Confidentiality)

Participant Revision of Statement:

- Alice's Purple Key (APUK) should no longer exist
- Alice's Red Key is shared with Bob and now becomes a shared key. (Transition from Asymmetric to Symmetric Key Cryptography)

Revision of Diagram:

- Erase Purple Key (APUK)
- The Red Key is now a shared symmetric key between Alice and Bob
- Participants explains that this is not how TLS works, but we will use this for the exercise
- Alice and Bob created the Red Key together

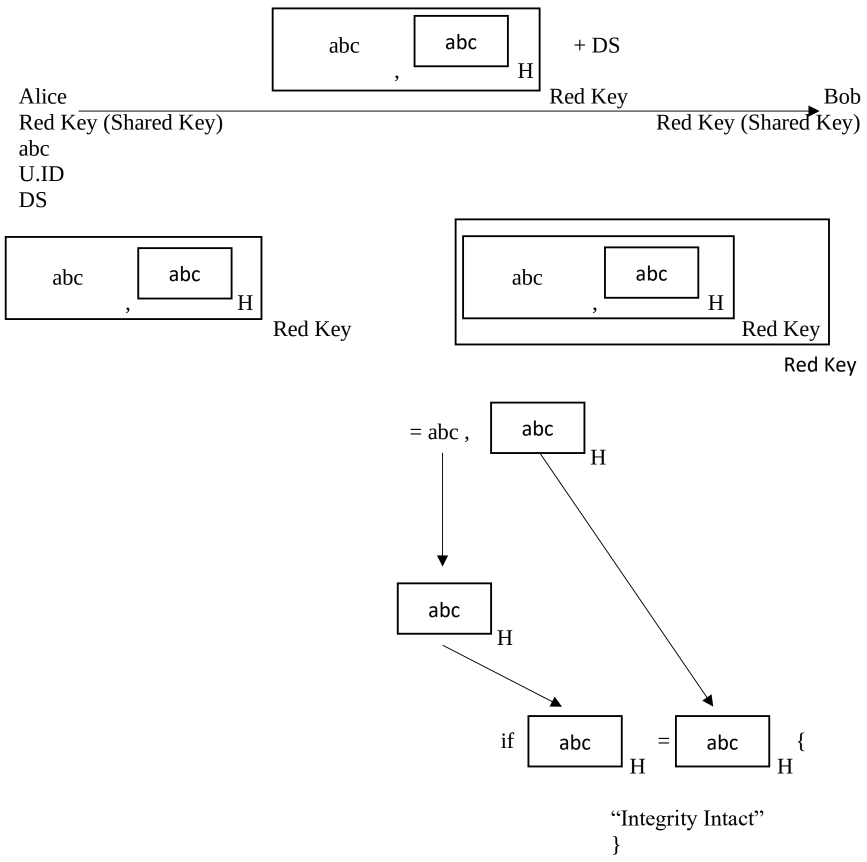


8. Bob decrypts the ciphertext with the Red Key

//

Participant 6

Condition: Integrity



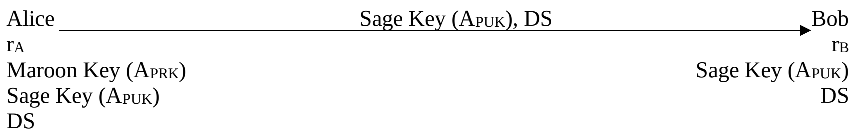
//

Participant 6

The entire process is known as the Data Exchange Process

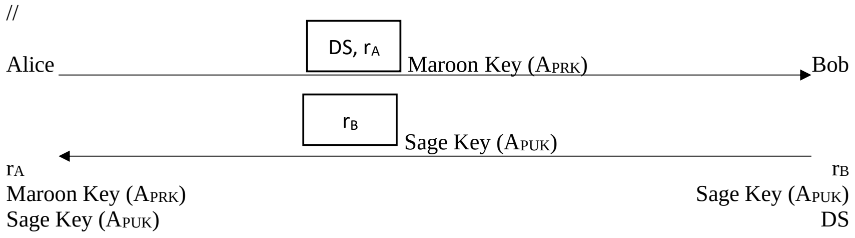
The process below is a prequel to the Data Exchange Process, known as the Key Exchange Process

- Key Generation
- Sharing of Red Key (Symmetric Key) between Alice and Bob



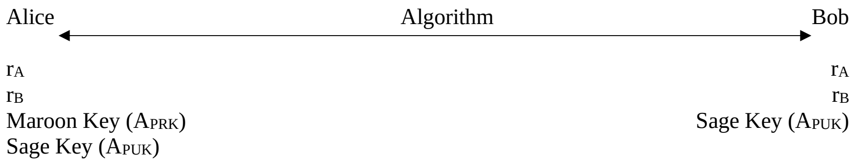
1. Alice generates a random number, r_A
2. Bob generates a random number, r_B
3. Alice sends Sage Key (A_{PUK}) and DS to Bob

//



4. Alice and Bob exchange r_A and r_B. Alice encrypts r_A with her private key: Maroon Key (A_{PRK}). Bob encrypts r_B with Alice's public key: Sage Key (A_{PUK}). Alice also encrypts DS with her private key.
5. Bob verifies Alice's DS

//



6. Alice and Bob agree upon an algorithm (Algorithm) for forming the symmetric Red Key.
7. Alice and Bob use a combination of r_A, r_B, and the Algorithm to form the symmetric Red Key

Alice	Bob
Red Key	Red Key

A.7 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 7

Participant 7

Initial Thought:

- RSA Encryption
- Or Symmetric Key Encryption

//

Condition 1: Confidentiality

Symmetric Key Generation achieved by Diffie-Hellman Key Exchange

1. Alice will have a secret key (x)
2. Bob will have a secret key (y)
3. g is a very large number and a public variable

g

Alice Bob

x y

4. Alice will compute g^x
5. Bob will compute g^y

g

Alice Bob

x y

g^x g^y

6. Alice and Bob exchange g^x and g^y

g

$$\begin{tikzcd} \text{Alice} \arrow[r, shift left, "g^x"] & \text{Bob} \arrow[l, shift left, "g^y"] \end{tikzcd}$$

x y

g^x g^y

Adversary cannot get the secret keys of Alice and Bob, x and y.
It is computationally infeasible to retrieve x and y from g^x and g^y.

7. Alice computes $(g^y)^x = g^{xy}$
8. Bob computes $(g^x)^y = g^{xy}$
9. Alice and Bob now have g^{xy}, the same symmetric key

$$\begin{tikzcd} \text{Alice} \arrow[r, leftrightarrow, "g^{xy}"] & \text{Bob} \end{tikzcd}$$

Participant 7

//

10. Now, both Alice and Bob have the symmetric key: g^{xy}, from now known as k.
11. We encrypt a message by:

$$m \oplus k$$

   a. message = ‘HELLO’
   b. Convert ‘HELLO’ into binary, B
   c. B $\oplus$ k = Ciphertext (Encryption)

$$\text{Alice} \xrightarrow{\text{Ciphertext}} \text{Bob}$$

12. Bob decrypts ciphertext by:

$$c \oplus k$$

Reasoning:

$$c = m \oplus k$$
$$m = m \oplus k \oplus k$$
$$m = c \oplus k$$

//

Condition 2: Authentication

We can use the RSA scheme for message authentication (Condition 3: Integrity) and entity authentication (Condition 2: Authentication).

The 3 stages of RSA:

- Key Generation
- Signing
- Verification

Key Generation:

1. Choose two very large prime numbers, p and q
2. Calculate product (n) of p and q, n = pq
3. $\phi n = (p - 1)(q - 1)$
4. Choose e randomly such that e is relatively prime to ϕn
5. Because e is relatively prime to ϕn, the inverse of e is d
6. ed is congruent with 1mod(ϕn)
7. e = Verification Key, held by Bob
8. d = Signing Key, held by Alice

Participant 7

Signing:

1. Take message m and apply hash function on it, $\{m\}_H$
2. Sign $\{m\}_H$ with the Signing Key (d), $(\{m\}_H)^d$
3. Signature can be labelled Sigma, $\Sigma = (\{m\}_H)^d$

//

Verification:

$$\text{Alice} \xrightarrow{\{m, \Sigma\}_k} \text{Bob}$$

e (Verification Key)
k (Symmetric Key)

$$\begin{aligned} &(\Sigma)^e \\ &= ((\{m\}_H)^d)^e \\ &=(\{m\}_H)^{ed} \\ &=\{m\}_H \end{aligned}$$

Rule:

- $ed = 1 + k\phi n$
  $ed = 1$

//

Condition 2: Authentication

Input $\boxed{\text{Verification Function}}$ Output

$(\Sigma)^e$

```
if (ed = 1){
    True
} else {
    False
}
```

True: Alice ID Authenticated
False: Mallory pretending to be Alice

Example of False: Mallory uses e’ instead of e. Then e’d != 1.

//

Participant 7

Condition 3: Integrity

1. $(\sum)^e$ should equal $\{m\}_H$
2. Take m and apply hash function, $\{m\}_H$
3. Compare hashes to check for tampering, $\{m\}_H = \{m\}_H$
    a. Hash function is a collision resistant function
    b. Two distinct inputs will not have the same value

A.8 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 8

Participant 8

Initial Thought:

- Concept of keys to help establish a secure connection
- There are 2 types of keys:
  - Symmetric Keys
  - Asymmetric Keys

//

Description of a Symmetric Key:

- The sharing of individual keys with individual people
- Alice and Bob want to communicate so they share one key between them

$$\begin{tikzcd}
\text{Alice} \arrow[r, leftrightarrow] & \text{Bob} \\
K & K
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Alice} \arrow[r, leftrightarrow, "\text{(Encryption)} \; \{m\}_K"] & \text{Bob} \\
K & K
\end{tikzcd}$$

$\{\{m\}_K\}_K$ (Decryption)
= m

Problem with Symmetric Key Connection:

- If Alice wants to communicate with 10 people, Alice has to store 10 keys
- Although storing 10 keys may be okay, storing 1000 keys to communicate with a 1000 people is a problem

//

Description of Asymmetric Keys:

- Each individual has 2 keys, a public key and a private key
- The public key is known to all
- The private key is known only to the individual
- A message encrypted with the private key can be decrypted with the public key
- A message encrypted with the public key can be decrypted with the private key
- These are the RSA Algorithm keys

//

Participant 8

Condition 1: Confidentiality

$$\begin{tikzcd}\text{Alice} \arrow[r] & \text{Bob}\end{tikzcd}$$

m
A_{PUK}
A_{PRK}
B_{PUK}

B_{PUK}
B_{PRK}

$\{\{m\}_{BPUK}\}_{BPRK}$
$= m$

Alice can be assured that only Bob can read the message.

Threat:

- Bob cannot be assured that the message came from Alice.

//

Condition 2: Authentication

$$\begin{tikzcd}\text{Alice} \arrow[r, "\{\{m\}_{APRK}\}_{BPUK}"] & \text{Bob}\end{tikzcd}$$

m
A_{PUK}
A_{PRK}
B_{PUK}

B_{PUK}
B_{PRK}
A_{PUK}

$\{\{\{m\}_{APRK}\}_{BPUK}\}_{BPRK}$
$= \{m\}_{APRK}$

$\{\{m\}_{APRK}\}_{APUK}$
$= m$

Bob can now be assured that the message came from Alice.

//

Condition 3: Integrity

Signatures ensures message cannot be tampered with during transmission.

The message cannot be tampered with because the adversary does not have the private keys.

Concept of Hashing from Blockchain can be used if needed.
- If data is manipulated, hash will not match with other peers' hash

Message cannot be read, let alone tampered with.

A.9 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 9

Participant 9

Initial Thought:

- Public/Private Key Pairs

Condition 1: Confidentiality

$$\begin{tikzcd} \text{Alice} \arrow[r, "\{m\}_{BPUK}"] & \text{Bob} \end{tikzcd}$$

m
A_{PRK}
A_{PUK}
B_{PUK}

B_{PRK}

$\{\{m\}_{BPUK}\}_{BPRK}\}$
$= m$

Eve cannot read the message because Eve does not have Bob’s private key.

//

Condition 2: Authentication

Initial Thought:

- Signatures

$$\begin{tikzcd} \text{Alice} \arrow[r, "\{\{m\}_{APRK}\}_{BPUK}"] & \text{Bob} \end{tikzcd}$$

m
A_{PRK}
A_{PUK}
B_{PUK}

B_{PRK}
B_{PUK}
A_{PUK}

$\{\{\{m\}_{APRK}\}_{BPUK}\}_{BPRK}$
$= \{m\}_{APRK}$

$\{\{m\}_{APRK}\}_{APUK}$
$= m$

//

Condition 3: Integrity

Initial Thought:

- Message cannot be tampered with

Testing system through Adversary:

- Participant questions how Eve can tamper with the message
- Eve has Alice and Bob’s public keys
- Eve does not have Alice and Bob’s private keys

Participant 9

- Participant concludes that Condition 3 is solved
- Participant questions how Mallory would get Alice’s private key in the first place

//

Thought:

- Finding hash of message to make sure Bob knows that the message is from Alice
- Participant does not think hashing is required

A.10 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 10

Participant 10

Initial Thought:

- Public/Private Keys (Mathematically Related Keys)
- Use some type of encryption standard

$$\begin{tikzcd} \text{Alice} \arrow[r, "\{m\}_{APRK}"] & \text{Bob} \end{tikzcd}$$

Alice: A_{PRK}, A_{PUK}

Bob: B_{PRK}, B_{PUK}, A_{PUK}

$\{\{m\}_{APRK}\}_{APUK}$
$= m$

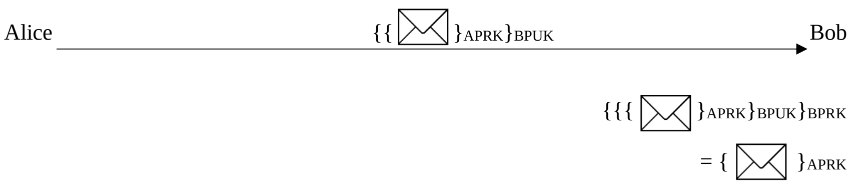
Threats:

- Anyone can read this message
- We know that it comes from Alice (Condition 2: Authentication)
- But this does not mean that only Bob can read the message

//
Condition 1: Confidentiality

Thought:

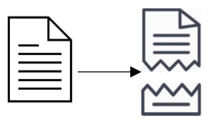
- We need to setup some kind of channel first



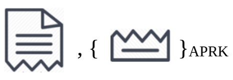
//

Revision (Condition 2: Authentication):

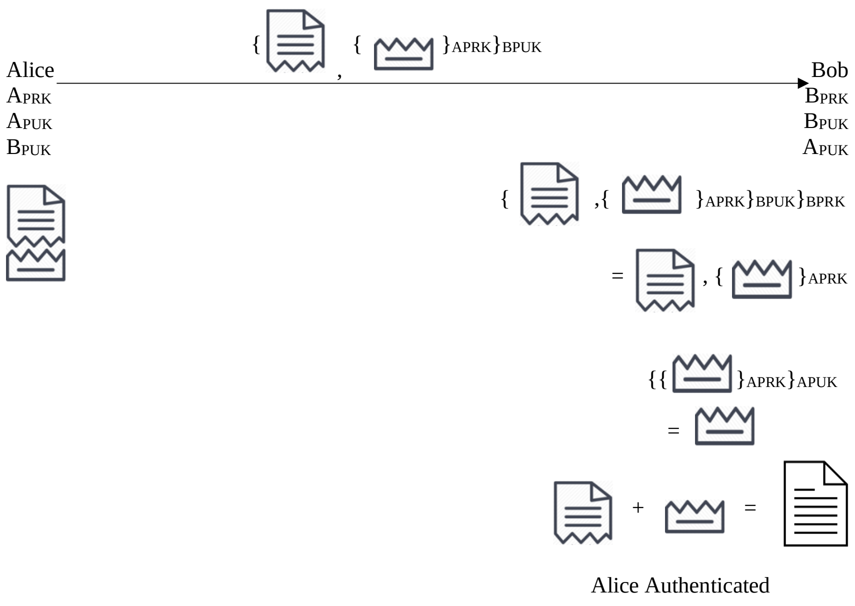
Alice



Alice encrypts half of the message with the intention of verifying her identity.



Participant 10



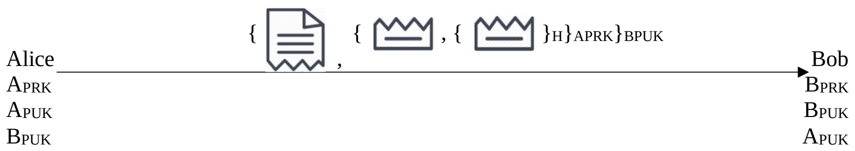
Participant only applies A_{PRK} to one half of the message to help improve the speed of the encryption.

//

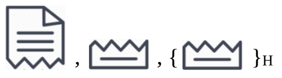
Condition 3: Integrity

Initial Thought:

- Hashing



First, decrypt using B_{PRK}
Second, decrypt using A_{PUK}
Now, you are left with the following:



Participant 10

Apply hash function to:

♛ = { ♛ }$_H$

Compare for Integrity:

{ ♛ }$_H$ = { ♛ }$_H$

A.11 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 11

Participant 11

Initial Thought:

- Two types of encryption:
  - Asymmetric
  - Symmetric

Description of Symmetric Key Encryption:

- “Here is a key to encrypt the message. The other person also has the key to decrypt the message”

//

Condition 1: Confidentiality

The task requires Asymmetric Key Encryption:

Alice	Bob
A_{PRK}	B_{PRK}
A_{PUK}	B_{PUK}

Participant attempts to recall process:

- Participant cannot remember whether public keys are exchanged beforehand or during the initial message

//

1. Exchange public keys with the person you are sending the message to

$$\begin{tikzcd} \text{Alice} \arrow[r, shift left, "A_{PUK}"] & \text{Bob} \arrow[l, shift left, "B_{PUK}"] \end{tikzcd}$$

A_{PRK}	B_{PRK}
A_{PUK}	B_{PUK}
B_{PUK}	A_{PUK}

Eve (MITM)

2. Encrypt message with Bob’s public key

Alice

$\{m\}_{BPUK}$

Participant 11

3. Send encrypted message to Bob

$$\begin{tikzcd}\text{Alice} \arrow[r, "\{m\}_{BPUK}"] & \text{Bob}\end{tikzcd}$$

A_{PRK}	B_{PRK}
A_{PUK}	B_{PUK}
B_{PUK}	A_{PUK}
m	

Participant introduced condition:

- Keep private key to yourself

Description of Public/Private Key (Recollection of Asymmetric Cryptography):

- If you encrypt a message with the public key, they can decrypt it with their private key

//

4. Bob decrypts the message with Bob's private key

$$\begin{tikzcd}\text{Alice} \arrow[r, "\{m\}_{BPUK}"] & \text{Bob}\end{tikzcd}$$

A_{PRK}	B_{PRK}
A_{PUK}	B_{PUK}
B_{PUK}	A_{PUK}
m	
	$\{\{m\}_{BPUK}\}_{BPRK} = m$

Eve (MITM)

Participant introduces conditions:

- No-one else should be able to decrypt the message
- Public/Private keys are based on very large prime numbers, and have an inverse nature

More details:

- Use of the term 'handshaking'

Participant attempts to recall the process of Symmetric Key Encryption for a secure connection:

- Only one key is used to both encrypt and decrypt
- The symmetric key is distributed by a totally different mechanism

Participant 11

- “USB Pen handed over” or “delivered in a briefcase”

//

Attacker Mindset:

- What could Eve do if she intercepted the message and got a hold of it?
  - Eve does not have the private key of the recipient (Bob)
  - Eve cannot decrypt the message
  - Eve cannot do anything with the message

Participant concludes that Condition 1: Confidentiality is satisfied.

//

Condition 2: Authentication

Initial Thought:

- If the message is not encrypted with Bob’s public key, then the message will not be decryptable with Bob’s private key
- This indicates that the message comes from someone who Bob has not given his public key to
- Assumption: The public key, by default, is not available to all. The owner of the public key makes the key available to others they want to talk to

Revised Assumption:

- Bob’s public key is available to all

//

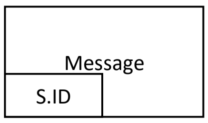
Thought:

- Sign the message with a signature
- JSON web tokens: not encrypted but encoded using Base64
- Signing mechanism used in JSON Tokens could be used here:
  - Alice can sign the message using her private key and apply a hash

//

Alice Bob

m =



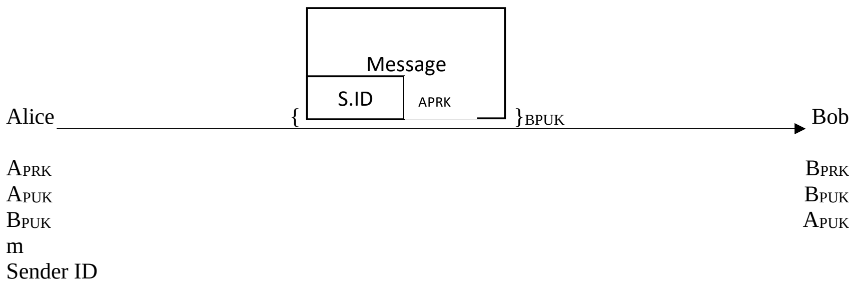
Sender ID = S.ID

$\{S.ID\}_{APRK}$ = Hash = Signature

Participant 11

Participant uses the term ‘hash’ to describe the output of signing the Sender’s ID with Alice’s Private Key.

5. Send signed message to Bob



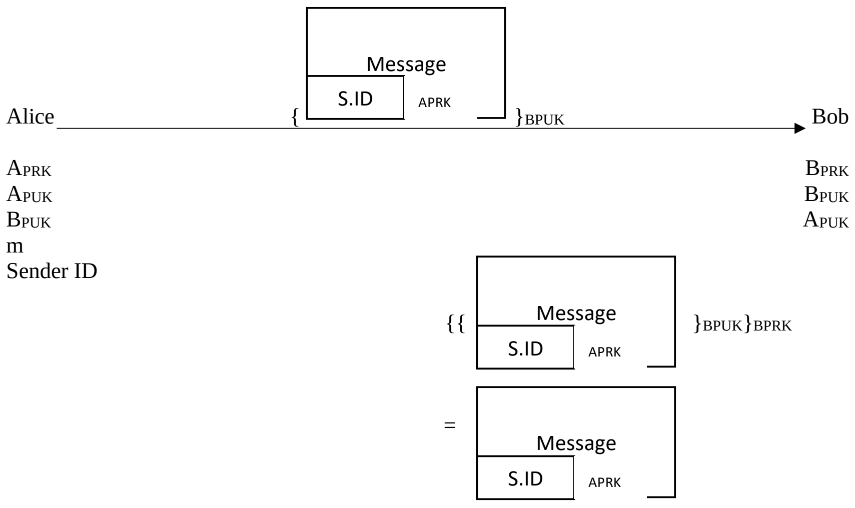
Alice applies the private key to the message to generate a ‘Hash’ or a ‘Checksum’.
This is the digital signature added as part of the message.

//

Verification:

- There would have to be some mechanism to verify the message
- Bob has Alice’s public key

6. Bob verifies the message with the expected sender’s (Alice’s) public key



Participant 11

$$\boxed{\text{S.ID}}_{APRK}$$

$$\{\boxed{\text{S.ID}}_{APRK}\}_{APUK}$$

= S.ID (Alice's Sender ID)

Participant concludes that Condition 2: Authentication is satisfied.

//

Condition 3: Integrity

Initial Thought:

- Process similar to Step 6
- Integrity based on how JSON Tokens are formed. E.g. JWT.io

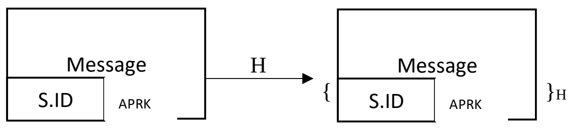
$$\text{Alice} \xrightarrow{\{\boxed{\begin{array}{c}\text{Message}\\ \boxed{\text{S.ID}}_{APRK}\end{array}},\ \{\boxed{\begin{array}{c}\text{Message}\\ \boxed{\text{S.ID}}_{APRK}\end{array}}\}_{H}\}_{BPUK}} \text{Bob}$$

A_{PRK} B_{PRK}
A_{PUK} B_{PUK}
B_{PUK} A_{PUK}
m
Sender ID

$$\{\{\boxed{\begin{array}{c}\text{Message}\\ \boxed{\text{S.ID}}_{APRK}\end{array}},\ \{\boxed{\begin{array}{c}\text{Message}\\ \boxed{\text{S.ID}}_{APRK}\end{array}}\}_{H}\}_{BPUK}\}_{BPRK}$$

$$= \boxed{\begin{array}{c}\text{Message}\\ \boxed{\text{S.ID}}_{APRK}\end{array}},\ \{\boxed{\begin{array}{c}\text{Message}\\ \boxed{\text{S.ID}}_{APRK}\end{array}}\}_{H}$$

Participant 11



Compare both of the hashes for Integrity

The hash is based on the context, not just the private key of the sender (Alice).

The message itself helps generate the Hash/Checksum. So if the message is changed then the Hash/Checksum will not match.

A.12 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 12

Participant 12

Condition 1: Confidentiality

Initial Thought:

- Since Alice is sending the message, she will generate two keys:
  - A public key: shared with both parties
  - A private key: kept solely by the sender

Alice	Bob
APUK	APUK
APRK	

Participant describes the features of the private key:

- The private key gives Alice the ability to see that Bob has her public key
- In order to create a public key, you need to have a private key
- The private key ensures that only a certain amount of people have access to the public key
- The private key can be used to cut off access and functionality of the public key if the public key falls into the wrong hands

Alice	Bob
APUK	APUK
APRK	

Holders of APUK
Alice
Bob

Alice	Bob
APUK	APUK
APRK	

Holders of APUK	
Name	Access
Alice	Granted
Bob	Granted
Eve	Denied

Eve
APUK

//

Participant 12

Bob receives the message and uses APUK to open it. Eve cannot read the message as she is not granted permission by Alice's private key (APRK).

$$\begin{tikzcd}
\text{Alice} \arrow[r, "\{m\}_{APUK}"] & \text{Bob}
\end{tikzcd}$$

APUK	APUK
APRK	
	$\{\{m\}_{APUK}\}_{APUK}$ = m

Note that Alice encrypts the message (m) using her public key (APUK). The use of APUK encrypt and decrypt by Alice and Bob, respectively, represents Symmetric Key Cryptography as opposed to the correct use of Public/Private keys.

//

Challenge:

- What happens if Eve intercepts APUK and pretends to be Bob?

Response (Revision of previous solution):

- A system is needed to validate the distinction cryptographically
- Participant references TCP/IP stack, SYN/ACK
- The system should notify Alice that Bob has received and opened the message
- When describing the notification, the participant refers to emails and read receipts

A Key ESCROW System:

- This system keeps a table of keys assigned to the people using the system
- The key represents the user's Unique ID
- Alice sends a message to the Key Escrow System to request Bob's Unique ID, with the intention of sending APUK to Bob

$$\begin{tikzcd}
\text{Alice} \arrow[r, shift left=2, "\text{“Request for Bob}_{U.ID}\text{”}"] & \text{K.E.S} \arrow[l, shift left=2, "\text{Bob}_{U.ID}"]
\end{tikzcd}$$

APUK	AliceU.ID
APRK	BobU.ID
BobU.ID	EveU.ID
	MalloryU.ID

- Alice sends APUK to Bob using Bob's Unique ID (BobU.ID) as a destination

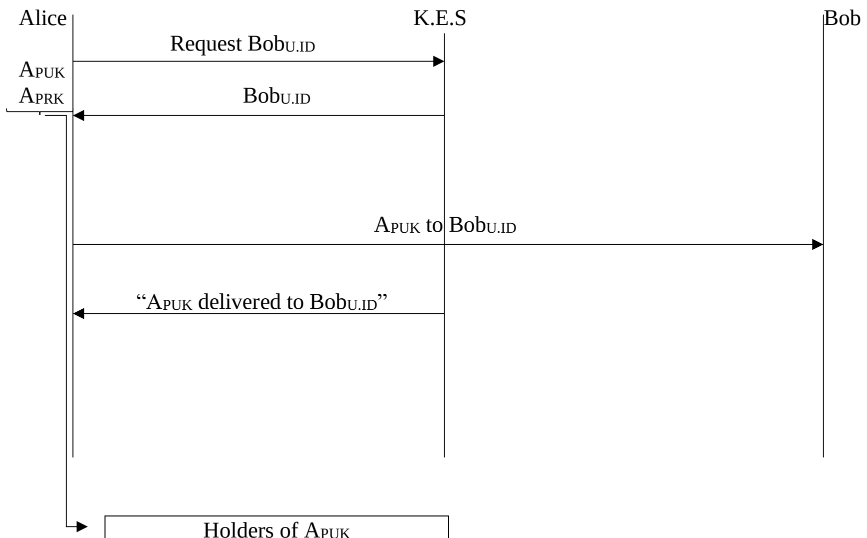
$$\begin{tikzcd}
\text{Alice} \arrow[r, "A_{PUK}\text{ to Bob}_{U.ID}"] & \text{Bob}
\end{tikzcd}$$

APUK	APUK
APRK	
BobU.ID	

- The Key Escrow System notifies Alice that A_{PUK} has been delivered to Bob, the expected destination

$$\begin{tikzcd}\text{K.E.S} \arrow[r, "\text{“}A_{PUK}\text{ delivered to }Bob_{U.ID}\text{”}"] & \text{Alice}\end{tikzcd}$$

Summary:



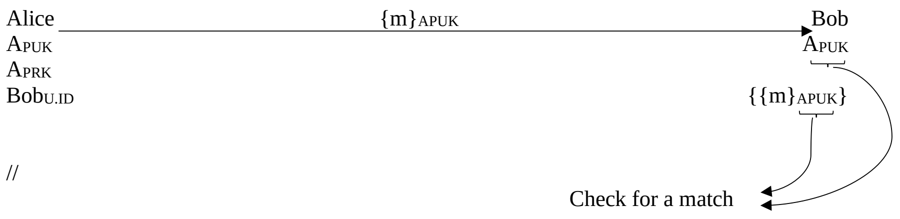
Holders of A_{PUK}	
Name	Access
Alice	Granted
Bob	Granted

//

Condition 2: Authentication

Initial Thought:

- I know that when sending a message, there is a signature attached to it
- Bob can match the signature with the key that Alice had initially sent him to verify that the message actually came from Alice



//

Participant 12

Condition 3: Integrity

Initial Thought:

- Alice can generate a hash using SHA-256, so that when Bob opens the messages, he can verify it using SHA-256 to validate the integrity of the message

Alice $\xrightarrow{\{m, \{m\}_{H}\}_{APUK}}$ Bob

A_{PUK}

A_{PRK}

$Bob_{U.ID}$

m

$\{m\}_{H\ (SHA\text{-}256)}$

A_{PUK}

1. Check A_{PUK}, A_{PUK}
2. $\{\{m, \{m\}_{H}\}_{APUK}\}_{APUK}$ $= m , \{m\}_{H}$

3. Apply H(SHA-256) to m $= \{m\}_{H\ (SHA\text{-}256)}$

4. Compare $\{m\}_{H}$ against $\{m\}_{H}$ for Integrity

A.13 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 13

Participant 13

Initial Thought:

- A secret key
- The secret key offers a symbolic representation of the message
- The Chart (secret key) is known only by Alice and Bob
- There would be a different chart for Alice and Charlie

Challenge:

- How does the chart work?

Response:

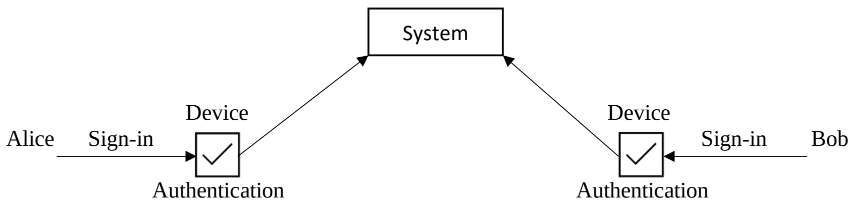
- A shifting alphabet (Caesar Cipher)
- E.g. Alice – Bob: ‘A’ = %
  Alice – Charlie: ‘A’ = *

Challenge:

How is the chart distributed between Alice and Bob?

Response:

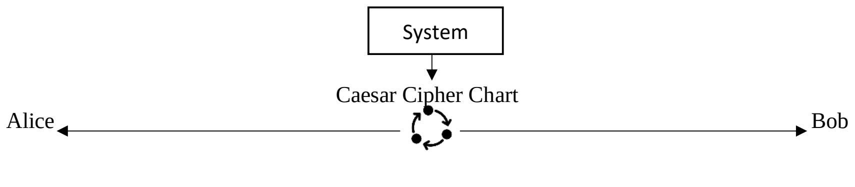
- An Authentication System (Condition 2: Authentication)
- Alice and Bob sign into a device using:
  - Gmail
  - User ID/Password
  - Digital Login



//

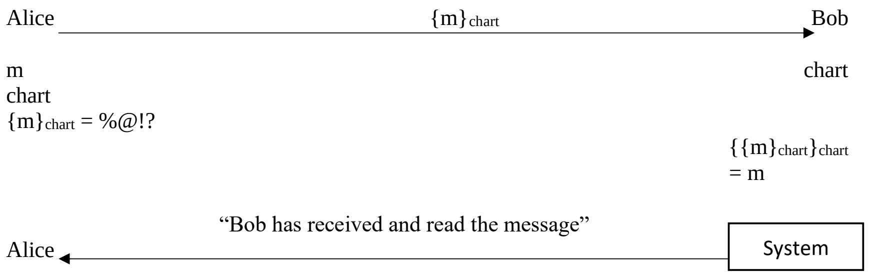
Condition 1: Confidentiality

System generates a chart only to be used by Alice and Bob.



Participant 13

//

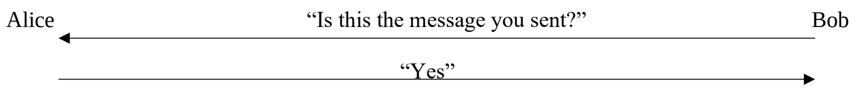


The system has a sync feature that notifies Alice the moment Bob has received and read the message.

//

Condition 3: Integrity

Since Alice and Bob are authenticated within the system, Bob can simply ask Alice if the message he received was the message she sent.

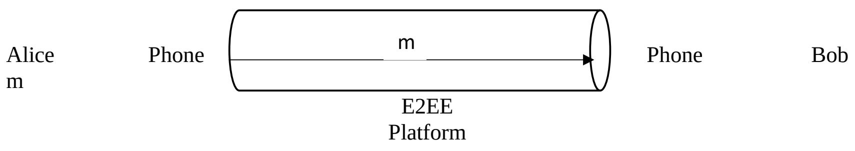


A.14 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 14

Participant 14

Initial Thought:

- There should be a medium between Alice and Bob
- Like using a phone when they are trying to have a chat
- Data going from Alice to Bob should be encrypted with End-2-End Encryption



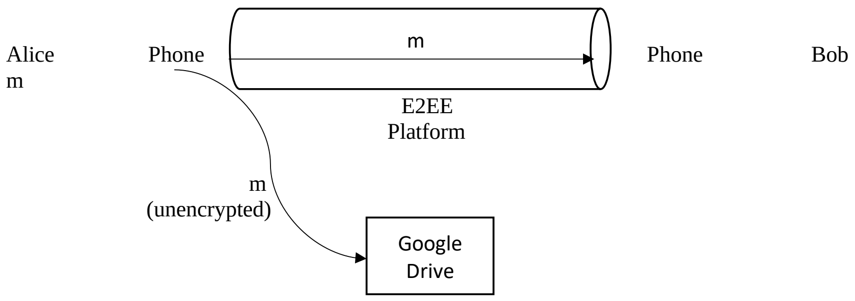
- Participant refers to WhatsApp
  - There should be an option to have an encrypted chat
  - There should be an option to backup to Google Drive

Challenge:

- What does encrypting the message mean?

Response (Threat Explained):

- Participant refers to Google Drive being used as a backup to WhatsApp
- If someone hacks Google Drive they can read the plaintext data:
  - Let's say that Alice is using WhatsApp
  - Her chats are stored as data on the app
  - On the fourth day, Alice decides to uninstall the app
  - By syncing Google Drive with WhatsApp, my data is backed up on Google Drive
  - In Google Drive, I can read the unencrypted messages
  - If Eve wants to hack Alice's WhatsApp
  - Eve can unlink Eve's WhatsApp account with Google Drive and form a new link with Alice's Google Drive account. Eve could try.
  - If Eve succeeds, Eve can read Alice's plaintext messages



Participant believes that Condition 1: Confidentiality is satisfied.

Participant 14

//

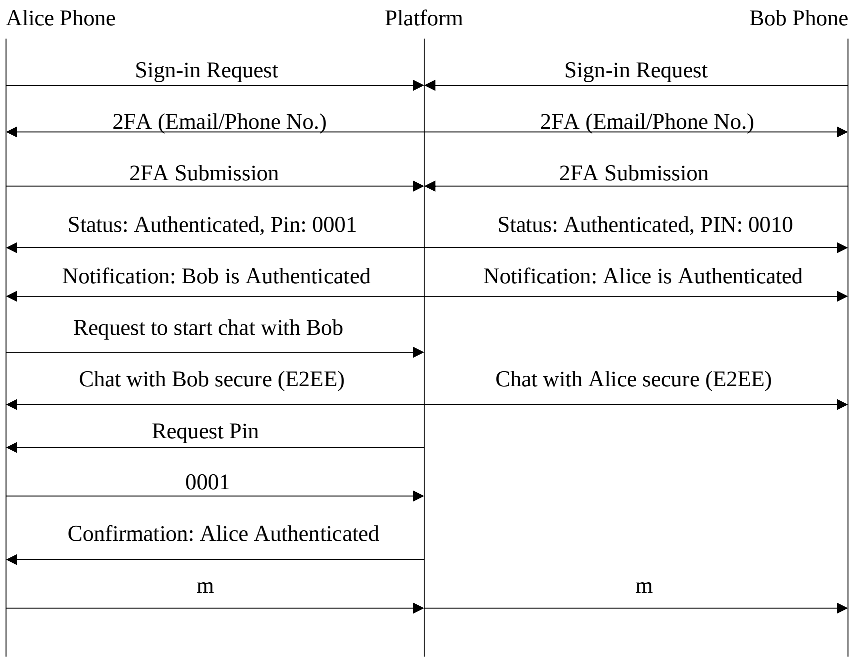
Condition 2: Authentication

Initial Thought:

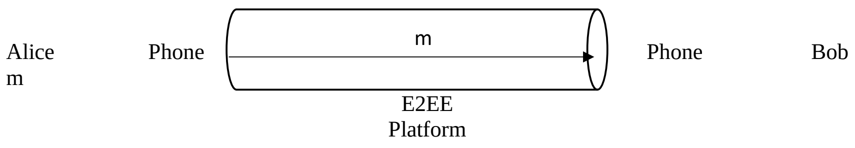
- There should be 2-Factor Authentication for Alice and Bob to access their messaging systems
- 2FA improves the security of the application
- However, people do not prefer 2FA because it takes a lot of time
- Make 2FA mandatory
  - Email Authentication
  - Phone Number Authentication
- Once 2FA is completed, both Alice and Bob are notified of each other’s authentication
- 2FA is not 100%, but it does improve the probability that the application will not break

Threat: Mallory

- Mallory pretends to be Alice
  - Alice performs 2FA
  - Alice leaves phone for a while
  - Mallory picks up the phone
  - Mallory wants to start a conversation with Bob
  - Phone requests a PIN (PIN is known only by Alice)


Alice Phone
Platform
Bob Phone
Sign-in Request
Sign-in Request
2FA (Email/Phone No.)
2FA (Email/Phone No.)
2FA Submission
2FA Submission
Status: Authenticated, Pin: 0001
Status: Authenticated, PIN: 0010
Notification: Bob is Authenticated
Notification: Alice is Authenticated
Request to start chat with Bob
Chat with Bob secure (E2EE)
Chat with Alice secure (E2EE)
Request Pin
0001
Confirmation: Alice Authenticated
m
m

Participant 14

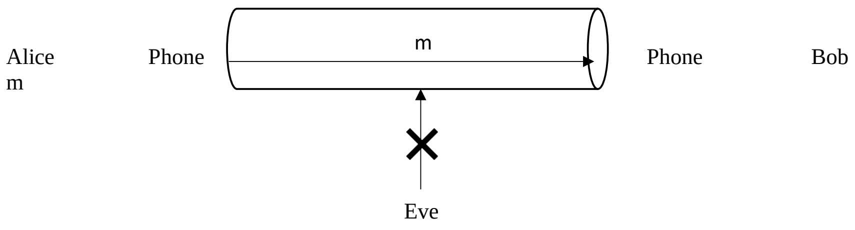


//

Condition 3: Integrity

Response:

- Achieved as it is an E2EE system



A.15 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 15

Participant 15

Condition 1: Confidentiality

Initial Thought:

- Alice encrypts the message and makes sure that only Bob knows how to decrypt the message
- Participant introduces condition: If Eve gets her hands on the message, she should not be able to decrypt it
- This way, Alice can be sure that only Bob can read the message

Challenge:

- Explain a system for sending the message “HELLO”

Response:

- Add 3 letters to ‘H’ to equal ‘K’
- Repeat for every letter
- Result: ‘KHOOR’
- Send ‘KHOOR’ to Bob
- Ask Bob to reverse the process

$$\begin{tikzcd}
\text{Alice} \arrow[r, "\text{‘KHOOR’}"] & \text{Bob}
\end{tikzcd}$$

m	KHOOR
m + 3	KHOOR - 3
HELLO = KHOOR	= HELLO

Challenge:

- How does Alice ask Bob to reverse the process by 3 letter?

Response:

- Add ‘3’ to the end of the message and send it to Bob

$$\begin{tikzcd}
\text{Alice} \arrow[r, "\text{‘KHOOR’, 3}"] & \text{Bob}
\end{tikzcd}$$

Challenge:

- What happens if Eve intercepts this message?

$$\begin{tikzcd}
\text{Alice} \arrow[r, dash] & \text{‘KHOOR’, 3} \arrow[r] \arrow[d] & \text{Bob} \\
& \text{Eve} &
\end{tikzcd}$$

Participant 15

Response:

- Send 2 messages:
    - First message is ‘3’
    - Second message is ‘KHOOR’

Challenge:

- What happens if Eve intercepts these two messages?

Response:

- We need a password known by only Alice and Bob
- We need an identifier for Bob

Threat:

- Even if Eve intercepts the message, Eve does not have the password to open it

$$\begin{tikzcd}\text{Alice} \arrow[r, leftrightarrow] & \text{Bob}\end{tikzcd}$$

Pass: 123 Pass: 123

Challenge:

- How does the password ‘123’ get shared between Alice and Bob?

Response:

- Alice and Bob agree upon a security question for which ‘123’ is the answer
- Bob has to know the answer before the message and question are sent to him

$$\begin{tikzcd}\text{Alice} \arrow[r, "\{\text{KHOOR}, 3\}_{Q}"] & \text{Bob}\end{tikzcd}$$

m Q
KHOOR A
3
Q
A

$\{\{\text{KHOOR}, 3\}_{Q}\}_{A}$
$= \text{KHOOR}, 3$

KHOOR - 3
= HELLO

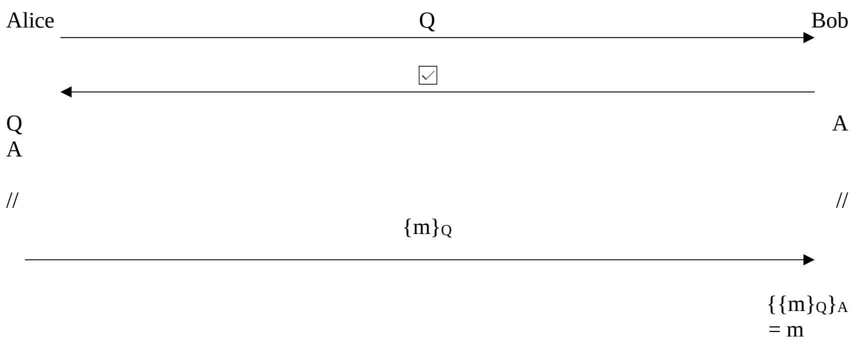
//

Participant 15

Condition 2: Authentication

Initial Thought:

- Since Bob and Alice agreed upon a Question and an Answer, Bob can be sure that if he successfully decrypts a message using his Answer, the message was sent by Alice



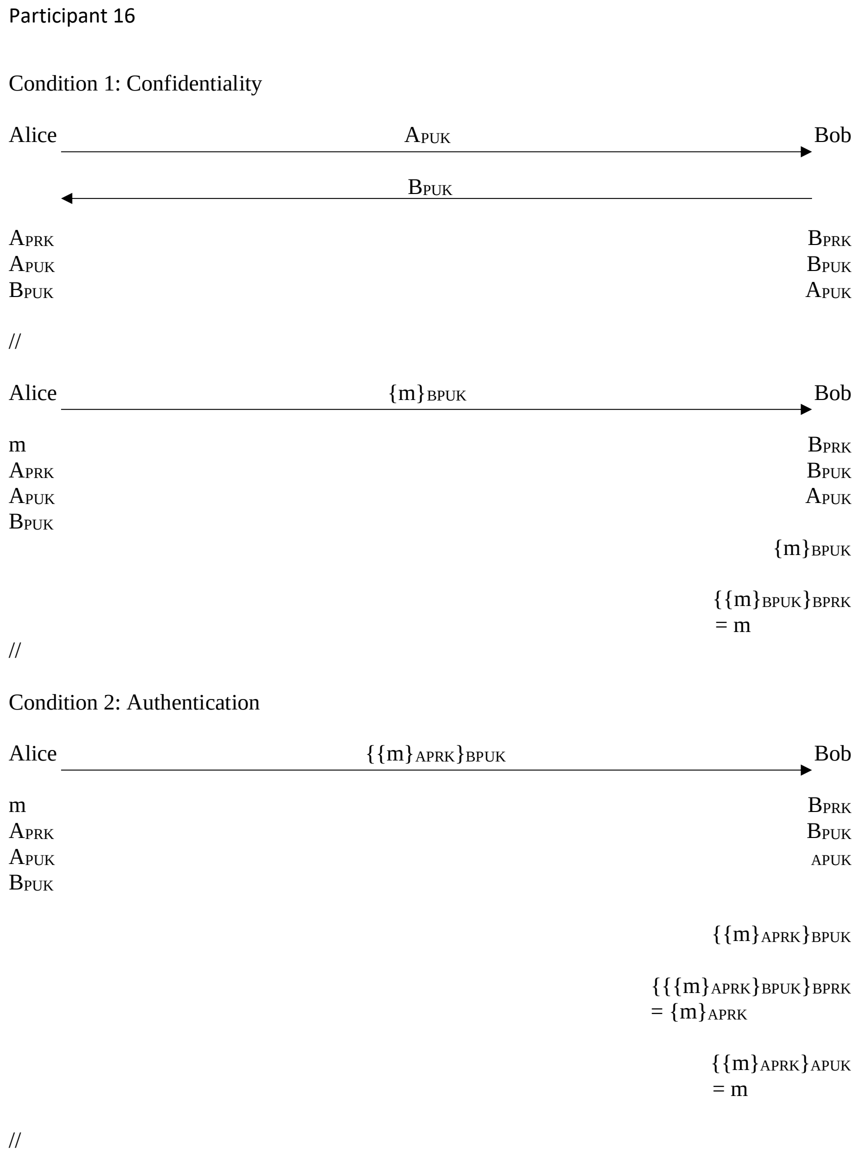
//

Condition 3: Integrity

Initial Thought:

- Eve cannot break through the Q &A system. Therefore, Eve cannot tamper with the message

A.16 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 16



Participant 16

Condition 1: Confidentiality

$$\begin{tikzcd}\text{Alice} \arrow[r, "A_{PUK}"] & \text{Bob} \arrow[l, bend left, "B_{PUK}"]\end{tikzcd}$$

A_{PRK}	B_{PRK}
A_{PUK}	B_{PUK}
B_{PUK}	A_{PUK}

//

$$\begin{tikzcd}\text{Alice} \arrow[r, "\{m\}_{BPUK}"] & \text{Bob}\end{tikzcd}$$

m	B_{PRK}
A_{PRK}	B_{PUK}
A_{PUK}	A_{PUK}
B_{PUK}	

$\{m\}_{BPUK}$

$\{\{m\}_{BPUK}\}_{BPRK}$
$= m$

//

Condition 2: Authentication

$$\begin{tikzcd}\text{Alice} \arrow[r, "\{\{m\}_{APRK}\}_{BPUK}"] & \text{Bob}\end{tikzcd}$$

m	B_{PRK}
A_{PRK}	B_{PUK}
A_{PUK}	$_{APUK}$
B_{PUK}	

$\{\{m\}_{APRK}\}_{BPUK}$

$\{\{\{m\}_{APRK}\}_{BPUK}\}_{BPRK}$
$= \{m\}_{APRK}$

$\{\{m\}_{APRK}\}_{APUK}$
$= m$

//

Participant 16

Condition 3: Integrity

$$\text{Alice} \xrightarrow{\{\{m, \{m\}_H\}_{APRK}\}_{BPUK}} \text{Bob}$$

m	B_{PRK}
$\{m\}_H$	B_{PUK}
A_{PRK}	A_{PUK}
A_{PUK}	
B_{PUK}	

$$\{\{\{m, \{m\}_H\}_{APRK}\}_{BPUK}\}_{BPRK}$$
$$= \{\{m, \{m\}_H\}_{APRK}$$

$$\{\{m, \{m\}_H\}_{APRK}\}_{APUK}$$

$$\begin{tikzcd}
 & {= m,} \arrow[dl, "H"'] & {\{m\}_H} \arrow[dd] \\
{\{m\}_H} \arrow[dr] & & \\
 & \text{Compare } \{m\}_H \text{ to } \{m\}_H\text{:} &
\end{tikzcd}$$

True: “Integrity Intact”
False: “Message has been tampered with”

A.17 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 17

Participant 17

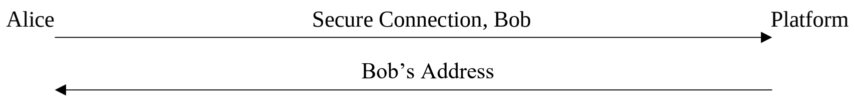
Condition 1: Confidentiality

Initial Thought:

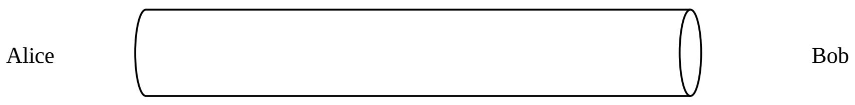
- Let's say the system is a real-time system, where Bob is waiting to receive the message

1. Establish a Secure Connection

- Alice requests the Platform for a secure connection with Bob
- The Platform gives Bob's address details to Alice

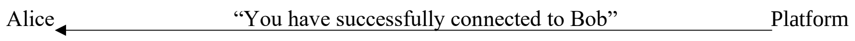


- Alice establishes the connection to Bob's address



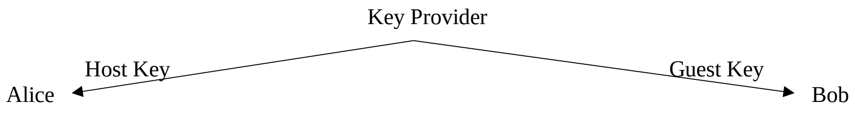
Established Secure Connection

- Alice gets acknowledgement from the Platform: "You have successfully connected to Bob"



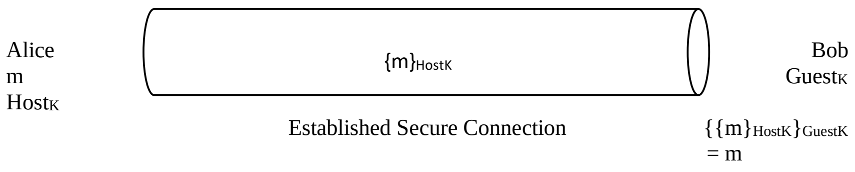
2. Encrypting the message

- The Encryption Key Sharing Mechanism will be a separate service than the messaging platform (Platform)
- Host and Guest Keys will be provided by this Encryption Key Sharing Provider (e.g. Microsoft or firms that sell keys for encryption)



- Alice encrypts the message using the Host Key
- Alice sends the encrypted message through the secure connection
- Bob receives the encrypted message and decrypts it using the Guest Key

Participant 17


Alice
m
$Host_K$
$\{m\}_{HostK}$
Established Secure Connection
Bob
$Guest_K$
$\{\{m\}_{HostK}\}_{GuestK} = m$

Threats:

- Even if Eve gets the message, she will not be able to decrypt it without the Guest Key
- Because a secure connection is established, the probability of a breach is low

Unique Scenario:

- If the message content is sensitive, you can put it in a USB stick and courier it to Bob
- Along with the message, Alice can provide a set of instructions:
    - Please install the USB into your machine
    - The message from the USB will seek out the Guest Key, which is already registered to Bob in the Key Provider system
    - Using the Guest Key, the message will convert to plaintext for you to read

//

Condition 2: Authentication

Initial Thought:

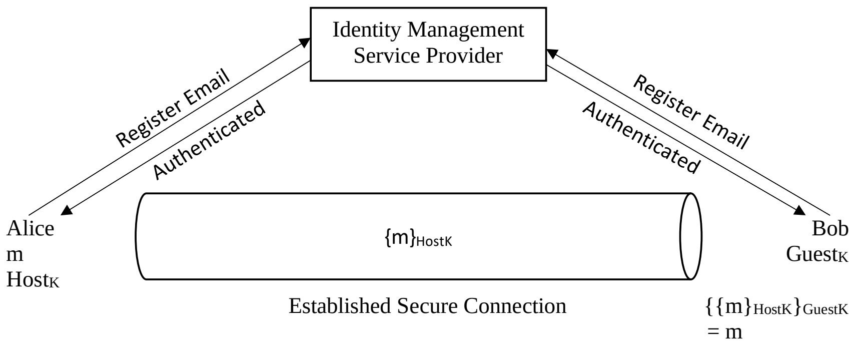
- We can identify a person by their email address:
- For example,
    - Alice has an email
    - Bob has an email
    - Alice can send a message to Bob as herself and it should be believed upon because it will be difficult for Eve to send an email from Alice’s mailbox
    - However, Eve can send an email from Eve@gmail.com and disguise it as Alice@gmail.com
    - If Bob does not look at this too carefully, he may think that the message is from Alice

//

Participant 17

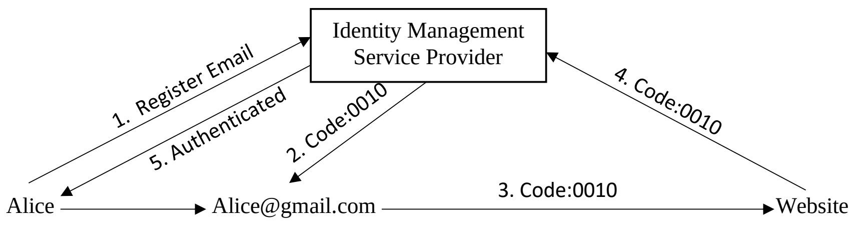
Identity Management Service Provider

- An Identity Management Service Provider can authenticate the email addresses of Alice and Bob and also authenticate the message, once Bob receives it



Revision of Identity Management Service Provider:

- The Identity Management Service Provider implements 2-Factor Authentication
- Upon receiving a registration requestion from Alice and Bob, the I.M.S.P sends a code to their registered email
- The I.M.S.P requests Alice and Bob to enter this code into the I.M.S.P website to authenticate them



Threat:

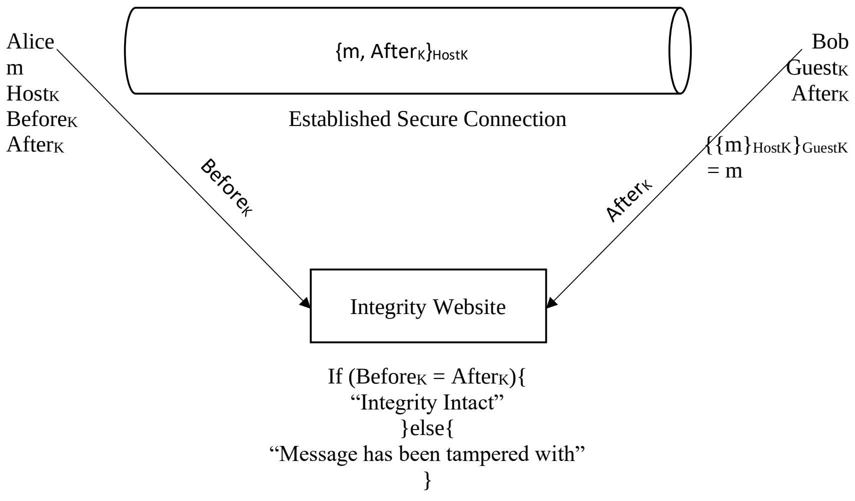
- If Eve pretends to be Alice, Eve would not have the code sent to Alice@gmail.com, and so Eve could not pretend to be Alice
- Another code inputted by Eve would be invalid
- The Identity Management Service Provider would become aware that someone (Eve) is trying to impersonate Alice
- The I.M.S.P lets Alice know that someone is trying to spoof her ID
- Bob will not get deceived by Eve

//

Participant 17

Condition 3: Integrity

- If we can be certain that this message has come from Alice then we can make a fair assumption that the content within the message has not been tampered with
- Using the size and content of the message, a Before and After Key are generated
- The After Key is sent to Bob along with the encrypted message
- Bob submits the After Key to a website
- Alice submits the Before Key to the same website
- The website compares the Before and After Key and checks for Integrity



A.18 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 18

Participant 18

Condition 1: Confidentiality

Initial Thought:

- Asymmetric Encryption
- Public/Private Key
- When Bob registers into a system, an ID is associated with Bob
- The system creates 2 keys for Bob, a public key and a private key
- The system shares the private key with Bob, and so nobody else has the key

$$\begin{tikzcd}
\text{Bob} \arrow[r, shift left, "\text{Registration Request}"] & \text{System} \arrow[l, shift left, "B_{PRK}"]
\end{tikzcd}$$

B_{PRK}

Bob:

ID: 1
Public Key
Private Key

- When Alice wants to send a message to Bob, Alice sends a message to the Server for Bob's Public Key
- Once Alice gets Bob's Public Key, Alice uses it to encrypt the message
- Alice sends the ciphertext to the Server

$$\begin{tikzcd}
\text{Alice} \arrow[r, shift left, "\text{Requests } B_{PUK}"] & \text{Server} \arrow[l, shift left, "B_{PUK}"] \\
\{m\}_{BPUK} \arrow[r, "\{m\}_{BPUK}"] & {}
\end{tikzcd}$$

- The Server sends the ciphertext to Bob
- Bob uses his private key (B_{PRK}) to decrypt the message

$$\begin{tikzcd}
\text{Server} \arrow[r, "\{m\}_{BPUK}"] & \text{Bob}
\end{tikzcd}$$

B_{PRK}

$\{\{m\}_{BPUK}\}_{BPRK}$
$= m$

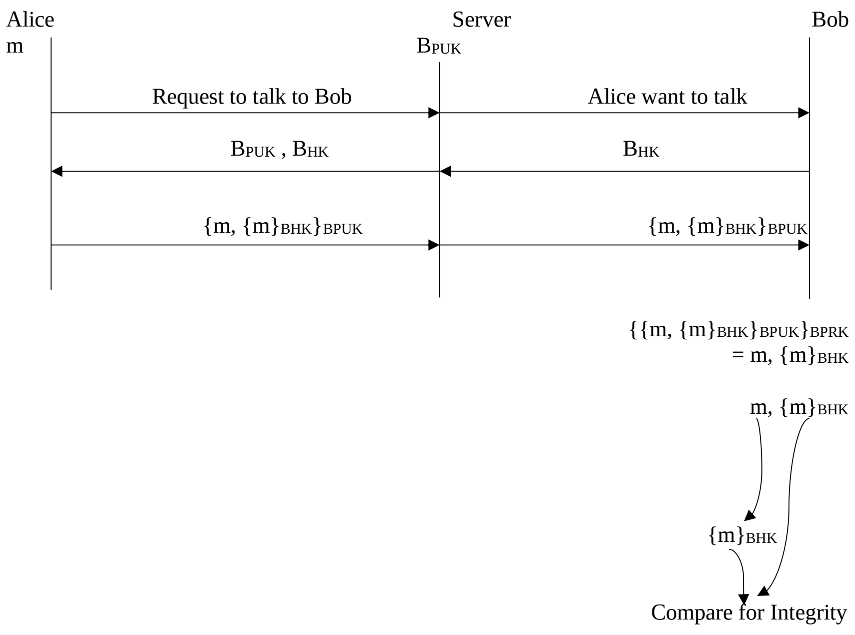
//

Participant 18

Condition 3: Integrity

Initial Thought:

- Hashing Key
- Bob has a private hashing key/hashing algorithm
- Bob shares this hashing key with Alice
- Alice sends a hash of the message using the hashing key along with the ciphertext
- Bob decrypts the ciphertext and applies his hashing key to the plaintext message, and then compares the two hashes for integrity



//

Participant 18

Condition 2: Authentication

Initial Thought:

- Because the Private Hash Key/Algorithm is only shared with Alice, Bob can assume that Alice sent him the message if his comparison of the two hashes returns TRUE

Challenge:

- Eve could have pretended to be Alice this whole time

Thought:

- 2-way communication
- Bob uses Alice’s Public Key to identify her
- Add Alice’s Private Key along with the encrypted message
- Bob will use Alice’s Public Key to check if the source is correct

A.19 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 19

Participant 19

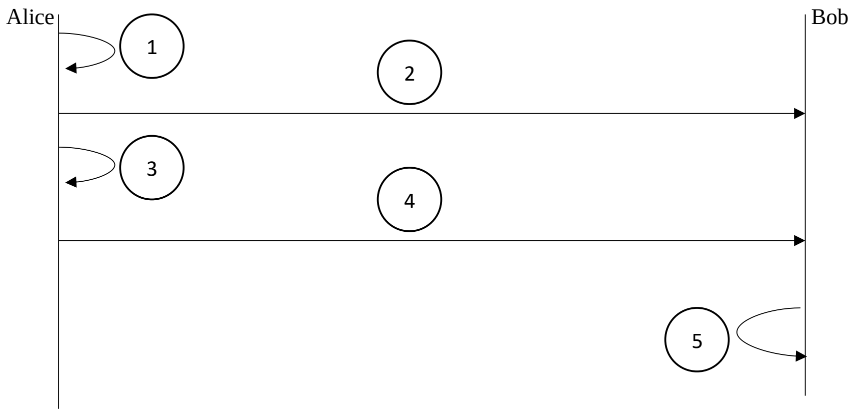
Condition 2: Authentication

Initial Thought:

- Making sure that the communication between Alice and Bob is secure
- Draw a sequence of events that take place between Alice and Bob
- Both Alice and Bob need to have a pair of public and private keys
- Alice needs to share the public key with Bob

Series of Events

1. Alice generates Public/Private keys
2. Alice shares her Public key (A_{PUK}) with Bob
3. Alice encrypts the message using her Private key
4. Alice sends the encrypted message to Bob
5. Bob decrypts the message using the Public key



//

Condition 1: Confidentiality

Challenge:

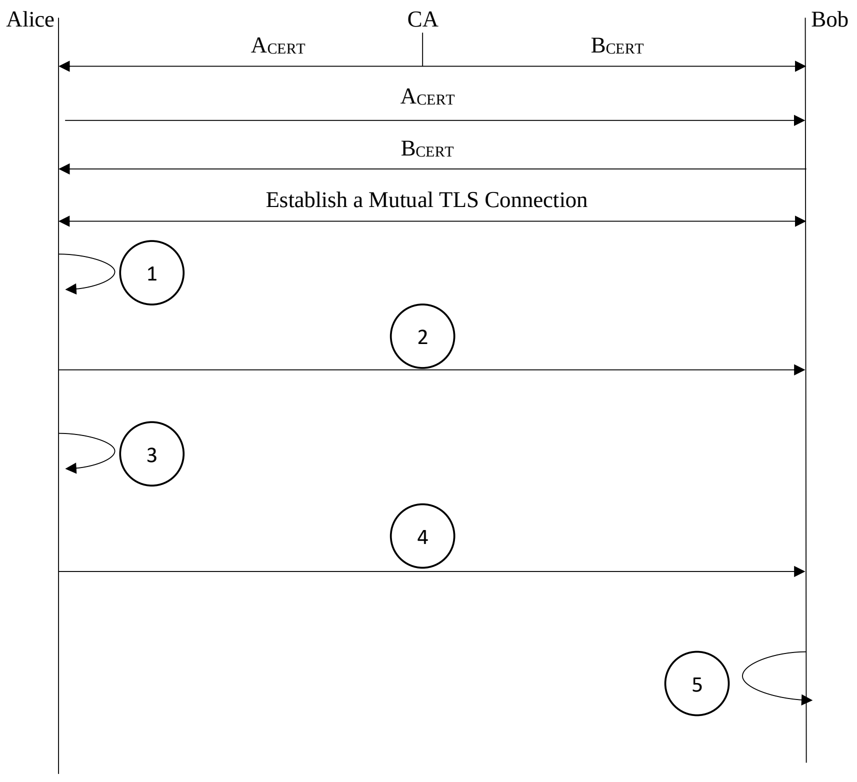
- How can Alice be sure that the message can only be opened by Bob?

Response:

- 2-way TLS
- Establish a mutual TLS connection between Alice and Bob
- Both of them can trust each other using certificates generated and distributed by a Certificate Authority
- We rely on a Certificate Authority to ensure that Alice is Alice and Bob is Bob

Participant 19

Diagram Revision:



Establish a Mutual TLS Connection:

- Both Alice and Bob will share each other’s certificates
- A trust authority will be creating the certificates for Alice and Bob. Alice and Bob add these certificates to their trust stores, and then move on to Step 1

//

Participant 19

Condition 3: Integrity

Initial Thought:

- If the message was tampered with, Bob would know straight away when he decrypts the message
- We can add some sort of hash, like MD5, to be sent by Alice
- MD5 is not a good one, people use SHA-256

Series of Events - Revision:

Revision made to Step 3.
Addition of new step: Step 6

1. Alice generates Public/Private keys
2. Alice shares her Public key (A_{PUK}) with Bob
3. Alice encrypts the message and a hash/checksum using her Private key
4. Alice sends the encrypted message to Bob
5. Bob decrypts the message using the Public key
6. Bob generates a hash/checksum of the plaintext message and compares it against the hash/checksum sent as part of the package

Additional Notes:

- It is the Server's responsibility to perform the overall task
- Purpose of Public/Private Keys:
    - Public Key:
        - You share it with someone you want to communicate with
        - It can be used to decrypt
    - Private Key:
        - Is only owned by you
        - It can be used to encrypt and decrypt

A.20 Chapter 4: Developer Mental Models of Public Key Cryptography: Participant 20

Participant 20

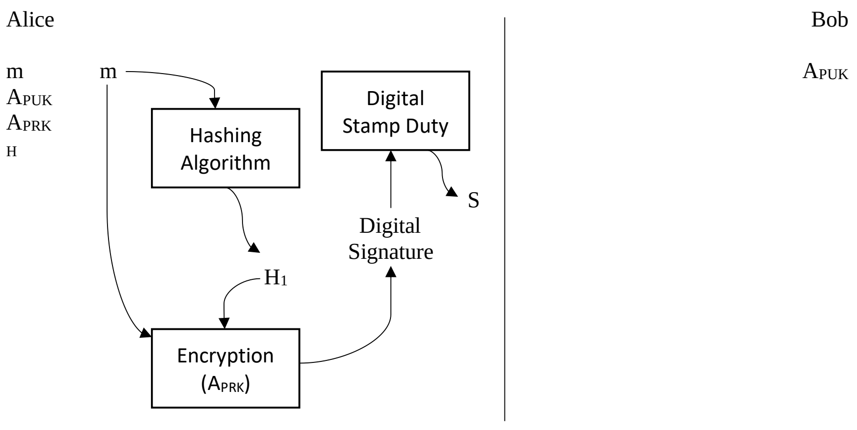
Condition 2: Authentication / Condition 3: Integrity

Initial Thought:

- Private Key
- Digital Signature
- Hashing Algorithm

Series of Events:

- Alice has the message
- Alice generates a hash of the message using the Hashing Algorithm, this is called the ‘hashed data’
- Encrypt the message and the hashed data using Alice’s Private Key (APRK)
- This encrypted package is Alice’s Digital Signature

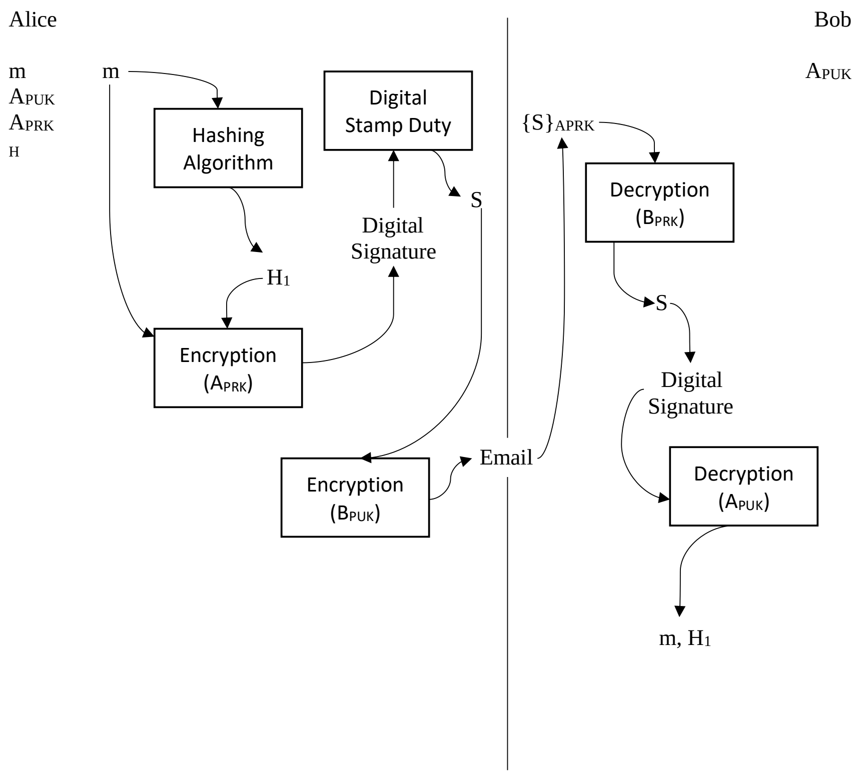


- The Digital Signature is Stamped through the Digital Stamp Duty to show that it has not been tampered with

//

Participant 20

Condition 1: Confidentiality



Series of Events:

- Alice encrypts the stamp with Bob's Public Key
- Alice sends the encrypted stamp to Bob via Email
- Bob decrypts the package using Bob's Private Key
- Bob checks the Digital Stamp Duty for signs of tampering
- Bob decrypts the Digital Signature using Alice's Public Key

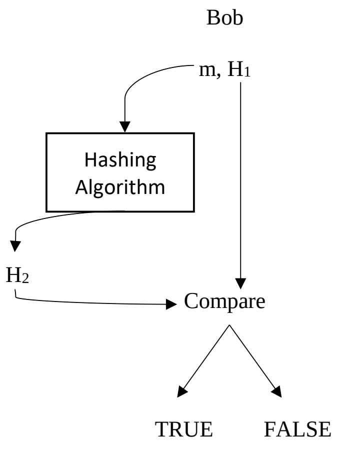
//

Participant 20

Condition 3: Integrity

Series of Events:

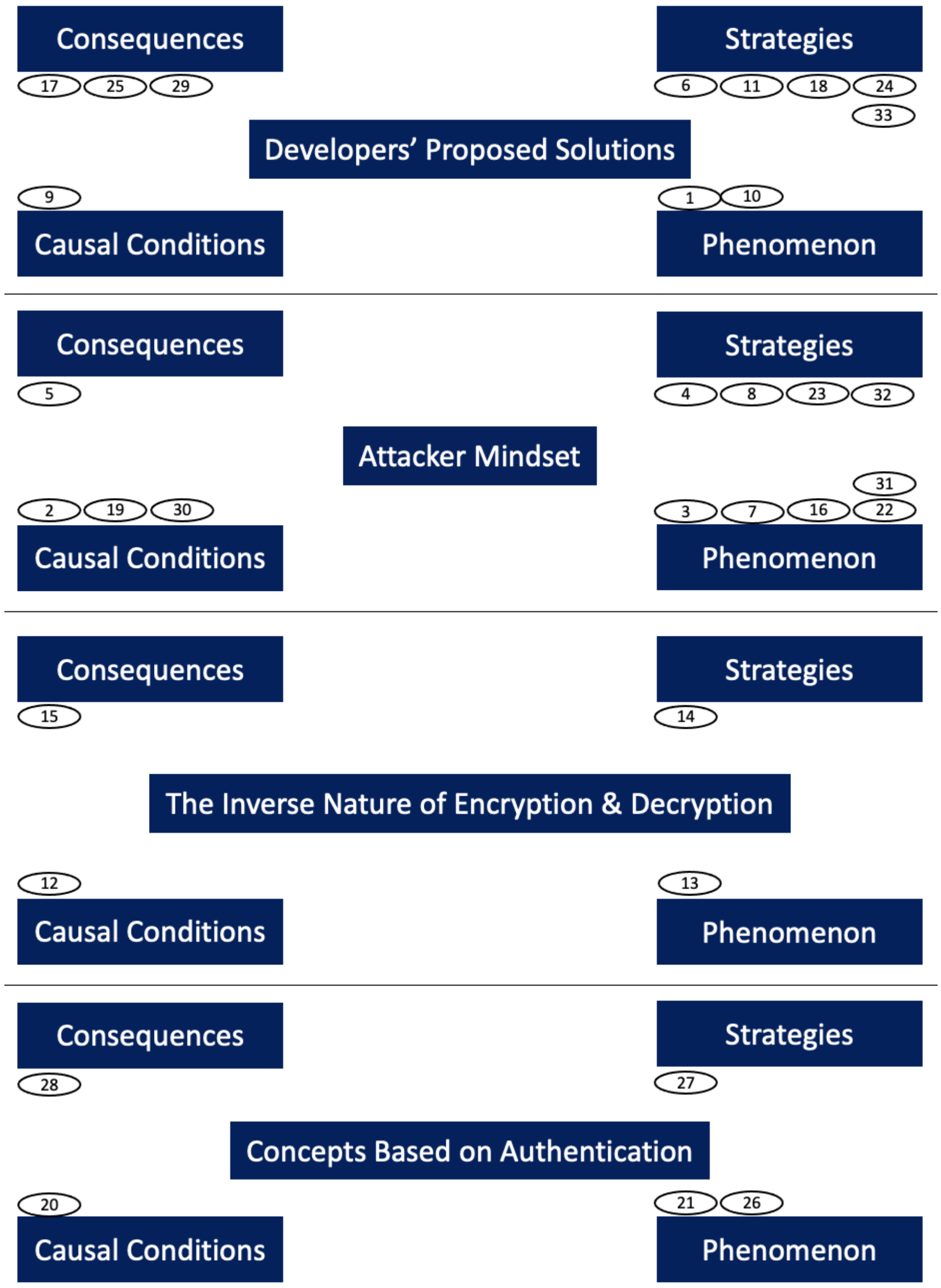
- Bob calculates the hash of the message, H_2
- Bob compares H_1 and H_2 for Integrity
- If they match then 'TRUE' else 'FALSE'



Additional Notes:

- Certificate Authorities can help address Condition 1: Confidentiality and Condition 2: Authentication
- Another way of checking for integrity: Bob can call Alice and say: "Did you send me this letter?"

A.21 Chapter 4: Developer Mental Models of Public Key Cryptography: Grounded Theory



A.22 Chapter 4: Developer Mental Models of Public Key Cryptography: Ethics Application

University of BRISTOL

Faculty of Engineering Research Ethics Committee

Upon completion this application form should be uploaded as an attachment, together with documents referred to in the application, to your online ethics submission. This form should be completed in conjunction with the guidance form.

University of BRISTOL

	Questions 1-12 **Contact Information and Study Details**
1.	**Title of the research:** Know Your Audience: Studying the relationship between the Developers' mental model and the API Designers' representation of Public Key Cryptography
2.	**Applicant details:** Student Name or Principal Investigator: Nikhil Patnaik Job or Course Title (UG or PG): PHD Student Contact number: 07931669457 Email: Nikhil.patnaik@hotmail.co.uk
3.	**Details of Supervisor (if applicant is a postgraduate or undergraduate student)** Name: Awais Rashid Title: Professor of Cyber Security Contact number: - Email: awais.rashid@bristol.ac.uk
4.	**Other investigator(s) involved, with job title:**
5.	**Source of funding:**
	NCSC Studentship
6.	**Start Date and Project Duration:**
	Start Date: 08/03/2021 Duration: 1 month
7.	**Where will the study take place?**
	Online Tools: Codeshare.io, Browserboard.com, Zoom.us

University of BRISTOL

8.	**Background and aims of the study:**
	Developers find cryptographic APIs challenging to use. This problem has initiated a rise in new usable cryptographic APIs with the sole purpose of providing an interface that is less prone to abuse and mistakes, while maintaining a strong level of security. However, the problem still remains, developers still struggle to use APIs, still misconfigure cryptographic functions, and are still unclear on the guarantees the API provides, but why? Attempts to answer these questions have been made by the usable security community, however, we take a different approach. Using a combination of the think-aloud protocol and a diagram exercise we elicit the developer's mental model while they attempt to solve the issues addressed by Public Key Cryptography. To clarify, we do not ask them how they perceive Public Key Cryptography. We ask them how they would solve the issues Public Key Cryptography solves without mentioning Public Key Cryptography. We do not ask the developer to tell us how they perceive Public Key Cryptography. We provide the developer key pairs for both Alice and Bob, then ask them how they would achieve the three conditions stated. We prime the developer, during the interview, by challenging their approach until they either reach the process of Public Key Cryptography which is generally deemed the standard, or they provide another solution and stop at what they think is a natural conclusion. We use a conventional approach to content analysis to help describe the developer's thought process and capture challenges faced through any hesitations they show. We summarize the coded data in a matrix to help identify basic patterns across subjects, perceptions, and challenges faced. To encourage the developer's thought process, we ask the developer to illustrate their thought process as they are solving the problem. The diagram exercise effectively logs the how the developer visualizes a solution to the problem and how they adapt their solution to attacks introduced by us to help prime to developer's thought process. The questions and attacks we raise help track the evolution of the diagram from initial thoughts to a final attempt. We compare the diagrams to correct implementation of Public Key Cryptography to check to developers who arrived at the correct implementation and negative cases, where developers did not come to the correct implementation. For negative cases, we analyze the diagram to see at which stages the developers started to feel challenged. Negative cases could also show developers who felt confident about their approach and found a different solution for the conditions of Public Key Cryptography. The diagram analysis combined with the audio analysis process allows us to further verify our elicitation of mental models. We also show the developer representations of Public Key Cryptography through the APIs of 2 cryptographic libraries; OpenSSL and LibSodium. Based on their ability to comprehend the different versions of code and their mental models we discover misalignments between the developer's mental model and the API Designers' conceptual model of Public Key Cryptography.

9.	**Outline the design of the study and list the procedures to which the participants will be subjected, the anticipated testing time and any treatments administered:**
	The Study: The study takes the form of an online experiment. We use a combination of the think-aloud protocol and a diagram exercise to elicit mental models from the developer. We create a study size of 25-50 participants. During the participant selection process, we select 10 cryptographic experts, 10 software developers, and 10 students studying cryptography at university level. We ask all the participants to take part in the diagram exercise first. We run the diagram exercise through sketchboard.io. Once the diagram exercise is complete, we start the second part of the study. We show the participant 2 versions of Public Key Cryptography. One version is written using the OpenSSL command line interface and the second version is written using Libsodium. OpenSSL presents a low-level (detailed) API, whereas Libsodium presents a higher-level API. We ask the participants to explain the code. In the exit interview, we show the participant a third implementation, written using OpenSSL's C-based API, and ask their thoughts on the representation. Background: Public Key Cryptography is a core component of many applications. Cryptographic libraries represent Public Key Cryptography through APIs of different abstraction levels. The representation is based on the API designer's conceptual model of how Public Key Cryptography should be performed. Although the API designer's conceptual model may be different, they ensure their conceptual model of Public Key Cryptography meets three conditions. Confidentiality: If Alice wants to send a message to Bob, Alice wants to be sure that only Bob can read the message. The message cannot be read by Eve. Authentication: If Bob receives a message he expected from Alice, Bob needs to know that Alice did indeed write the message and that it was not written by Mallory pretending to be Alice. Integrity: After receiving a message from Alice, Bob needs to be sure that the message has not been tampered with in transmission. These conditions are addressed by Public Key Cryptography. However, Public Key Cryptography is presented at varying levels of abstraction through different cryptographic APIs by API designers. We do not ask the developer to tell us how they perceive Public Key Cryptography. We provide the developer key pairs for both Alice and Bob, then ask them how they would achieve the three conditions stated. We prime the developer, during the interview, by challenging their approach until they either reach the process of Public Key Cryptography which is generally deemed correct, or they provide another solution and stop at what they think is a natural conclusion. We then show the developer how Public Key Cryptography is represented through 2 cryptographic APIs. The developer is asked to read through the code and provide their understanding of the API designer's conceptual model. Finally, we conduct an exit interview with the developer. We ask for the developer's opinion about the task they completed and the different versions of code they read through. We also adopted the System Usability Scale (SUS) to help quantify the developer's view on usability for the cryptographic API implementations. We record their demographics and programming experience. After the developer provides their understanding of the API designer's conceptual model, we are interested to see had a preference or understood one conceptual model better than others. If they

found a version difficult to comprehend, we asked what exactly the challenges are?

Data Analysis:

The interview consists of a diagram exercise and a comprehension study. We screen record the diagram exercise and the audio of each interview is also recorded and transcribed.

Audio Analysis:

We use a conventional approach to content analysis to help describe the developer's thought process, and capture challenges faced through any hesitations they show.

The transcriptions are jointly coded by two authors. Any disagreements while reviewing the transcripts were resolved through discussion.
We code the records based on commonalities of perception found in the study group, using a bottom-up approach to avoid preconceived categories and allow the categories to flow from the data itself.

We summarize the coded data in a matrix to help identify basic patterns across subjects, perceptions, and challenges faced.

Based on the initial summary matrices, we identify patterns in the way developers talked about solving the conditions of Public Key Cryptography, paying special attention to word choices, metaphors used, and explicit statements. We look for themes in which developers differ in their perceptions from the correct implementation of Public Key Cryptography (negative case analysis). These themes serve as the foundation upon which the mental models form.

We make a more detailed matrix in which we clustered developers who show similar results. This detailed matrix helped identify the mental models we encountered. To help verify the model descriptions, we revisit the data to check if the model description accurately represented the descriptions provided by the developers. While comparing the mental model description to the developer descriptions, we looked for contradictions and negative cases, allowing us to update the models with new insights. This was an iterative process; we checked for accurate representation until we found the model descriptions to accurately represent the developer's perceptions.

Finally, we present the developer with 2 representations of Public Key Cryptography through the selected cryptographic APIs. We ask the developer to read and attempt to comprehend the code. We show the participant a third implementation of Public Key Cryptography through the OpenSSL C-based API. We analyze their interpretation of the code. The developer may use certain words or metaphors to help explain the code. These metaphors may be seen as abstractions. The developer may show signs of hesitation which we class as challenges and indicate points of misalignment between the developer's mental model and the API Designer's conceptual model. Based on the abstraction level of the cryptographic API, some representations may show negative results through confusion, some may be preferred. The developer may lean more towards one representation than another. The relationship is entirely based on the relationship between the developer's existing perceptions and their ability to comprehend and adapt to the cryptographic APIs.

Diagram Analysis:

To encourage the developer's thought process we ask the developer to illustrate their thought process as they are solving the problem. The diagram exercise effectively logs the how the developer visualizes a solution to the problem and how they adapt their solution to attacks introduced by us to help prime to developer's thought process. The questions and attacks we raise help track the evolution of the diagram from initial thoughts to a final attempt.

University of BRISTOL

	We compare the diagrams to correct implementation of Public Key Cryptography to check to developers who arrived at the correct implementation and negative cases, where developers did not come to the correct implementation. For negative cases, we analyze the diagram to see at which stages the developers started to feel challenged. Negative cases could also show developers who felt confident about their approach and found a different solution for the conditions of Public Key Cryptography. The diagram analysis combined with the audio analysis process allows us to further verify our elicitation of mental models.
10.	**Does your study involve the collection or use of any human tissue or exudate? If yes, what is the material to be collected?**.
	Yes ☐ No ☒
	If yes, please explain:
10a.	**If you have answered 'yes' to Q10, has confirmation been obtained from your Departmental Human**

	Tissue Act Advisor that collection and storage of this material will be undertaken under an appropriate licence?
	Yes ☐ No ☐
11.	**Will the research involve working with animals?**
	Yes ☐ No ☒
	If yes, please identify how you will address any animal welfare issues and whether you have undertaken ethical review elsewhere (e.g. zoo or national park authorities). Please also see the relevant guidance.
12.	**Has this study been subjected to peer review?**
	Yes ☒ No ☐

University of BRISTOL

	Questions 13-22 **Recruitment and Informed Consent**
13.	**Who will be recruited to participate in this study?**
	We will recruit a combination of professional software developers, and students of cryptography at university level.
14.	**Are there any potential participants who will be excluded? If so, what are the exclusion criteria?**
	We require that participants self-report their level of understanding of cryptography and software development.
15.	**How many participants will be recruited?**
	We are aiming for 25 to 50 participants
16.	**How will the participants be recruited?**
	NCSC provide a service for connecting PhD students with software companies they know. I hope to recruit computer science students as well from the University of Bristol.
17.	**How will informed consent be obtained from all participants or their parents/guardians prior to individuals entering the research study?**
	We will ask for it as part of the experiment.
18.	**How long will potential participants have to decide whether to give consent?**
	They can give their consent before the experiment, but if they do not complete the survey, we will not record their responses. It will not be possible to remove responses after the survey is complete (due to anonymisation)
19.	**Will participants be kept informed of new information that becomes available during the study which may influence their continued participation**?
	We do not expect new information to appear for the duration of the experiment

University of BRISTOL

20.	**Will the study involve actively deceiving, or withholding information from, the participants?**
	Yes ☐ No ☒
	If YES, explain why it is necessary to use deception and state how you will ensure that the participants are provided with sufficient information at the earliest stage, and how you intend to ameliorate possible distress caused by the deception, including a plan for subject debriefing.
21.	**Will participants be made aware that they can withdraw from the study at any time without having to give a reason for doing so?**
	If the participant chooses to withdraw before or during the experiment, their responses will not be recorded. However, once the experiment is complete, their cannot withdraw your responses as they will be anonymised.
22.	**Describe potential risks (physical, psychological, legal, social) arising from these procedures:**
	None.
22b.	**Is there likely to be any risk to the investigator during this study?**
	Yes ☐ No ☒
	If yes, please explain how this will be minimised
22c.	**Is there likely to be any risk eg. legal, adverse publicity, to the UoB?**
	Yes ☐ No ☒
	If yes, please explain

	Questions 23-32 **Outcomes and Data Protection**
23.	**How will participants be informed about the outcome of the study?**
	We can let them know through email
24.	**How will the results of the study be disseminated and reported?**
	Through publication (we are targeting a peer-reviewed conference, or journal).
25.	**Is any payment other than reimbursement of expenses to be made to participants?**
	Yes ☒ No ☐
	A payment of a £20 Amazon gift voucher will be given to gather participants as part of the recruitment process
26.	**Will personal data, beyond that recorded on the consent form, be used in the research?**
	No.
27.	**Will the participants be audio-taped or video-taped?**
	Yes
28.	**What arrangements have been put in place to ensure confidentiality and security of data gathered in the study? Will the data be stored in hard copy or electronically, and where will it be held?**
	It will be stored on an encrypted filesystem in one drive. The survey itself will be delivered through Codeshare.io and Browserboard.com. We intend on using Zoom.us as an alternative method if there are any issues with Codeshare.io or Browserboard.com on the day. The survey will be deleted once the study is complete.
29.	**Has this proposal been seen by or submitted to another ethics committee?**
	No.

30.	**Do any of the investigators have any actual or potential conflict of interest in this study?**
	No.
31.	**Is there any other relevant information you would like to make known to the committee?**
	No.
32.	**How will the data be made available at the end of the project?** You must declare your level of access, see Data Access appendix
	Registration required.
33.	**Have you read and understood the guidelines for completing this form (see last page)?**
	Yes ☒ No ☐

Appendices

Informed Consent

Obtaining informed consent from parents does not obviate the need to obtain informed consent or assent from children participating in research. Assent means that the child shows some form of agreement to participate in the research without necessarily comprehending the nature of the research sufficiently to give full informed consent. Investigators working with infants should take special effort to explain the research to the parents and be especially sensitive to any indication of discomfort or avoidance in the infant.

It is good practice to ask participants on the consent form to confirm their consent to keep and make use of the data they have contributed. This allows someone, who for example becomes unhappy about their participation in the research, to prevent their data being used.

The researcher should keep signed copies of consent forms securely and separately from the research data.

For a questionnaire study, the researchers should consider if the questionnaires can be returned anonymously, in which case a consent form may not be necessary since consent is implied by the subject choosing to participate in the study. Under these circumstances, an information sheet is still required.

Data Access

Research funders and publishers increasingly require researchers to find a way to provide access to their research data, even if that data initially includes personal information.

The University of Bristol requires you to assign an expected access level to your research data, your selection will be checked and signed off by the Ethics Committee. If you intend to create multiple datasets with different anticipated access levels you should select the most restrictive access level you expect to use. The four access levels are:

•Open – my data can be made openly available through a data repository
•Registration required – my data should only be available to bona fide researchers, on request
•Controlled – any access requests for my data should be referred to committee for review on a case-by-case basis
•Closed – my data should not available for sharing

If, during the course of your research, you believe that your nominated access level will no longer be appropriate you should inform your Faculty Ethics Officer.

You must also ensure that you get the appropriate level of consent from participants at the start of the project to allow for onward use. If you need more information about this please see the guidance on sensitive data http://data.bris.ac.uk/research/storage-and-security/sensitive-data/ or contact data-bris@bristol.ac.uk
Guidance on access levels

Open – this level can be assigned where consent has been given by participants to make their anonymised data publicly available through a repository, in addition the risk assessment of re-identification of this anonymised data has been classed as low. These data sets can be made openly available through data repositories, including the Bristol Research Data Repository.

Registration required – this level can be assigned where consent has been given by participants to make their anonymised data available to bona fide researchers on request, within the terms of participant consent and the risk assessment of re-identification of the anonymised data is low. If the data is deposited with the University of Bristol Research Data Repository requests will be facilitated by the Research Data Service.

Controlled – this covers cases where historical consent for sharing is very limited and/or the risk assessment of re-identification is classed as medium to high. If the data is deposited with the University of Bristol Research Data Repository the Research Data Service will forward on requests to a Data Access Committee who will work with you as the PI to decide if/what data is appropriate to be made available.

Closed – this covers data that is not available for sharing (except by regulators) because of ethical, IPR, prior exclusive agreements or other constraints. This should only be assigned if you have got prior agreement from the funder that they are willing to allow the data to be completely closed.

University of BRISTOL

Before submitting this form, please refer to the checklist below.
(Do NOT include a copy of this checklist with your application)

Checklist

In assessing all applications, the Faculty Committee for Ethics will ask the following questions:

1. Do the likely benefits of the research outweigh the risks (if any) to the participants?

2. Are there possible risks to participants greater than they would normally encounter in their life outside research? If so, are adequate safeguards in place to minimise any harm?

3. Are there possible risks to investigators?

4. What degree of discomfort, distress or deception, if any, is foreseen?

5. Is the study adequately supervised and is the principal supervisor responsible for the project clearly identified, adequately qualified and experienced?

6. Are appropriate procedures (e.g. information sheet) in place for informing participants about the research study?

7. Are there proper procedures for obtaining consent from the participants or, where necessary, their parents or guardians?

8. Please attach (where appropriate)
   - ➢ Recruitment adverts / messages / forms
   - ➢ Information sheet / transcript
   - ➢ Consent form
   - ➢ Debriefing sheet / transcript
   - ➢ Questionnaire
   - ➢ Any other relevant material (e.g. an unpublished questionnaire enquiring about possibly sensitive topics or collecting personal data).

Links to useful guidelines concerning ethics of research involving human participants

ESRC Research Ethics Framework
http://www.esrc.ac.uk/ESRCInfoCentre/Images/ESRC_Re_Ethics_Frame_tcm6-11291.pdf#search='esrc%20research%20ethics%20framework

National Research Ethics Service (NRES)
http://www.nres.npsa.nhs.uk/

Medical Research Council Guidelines on Good Research Practice
http://www.mrc.ac.uk/pdf-good_research_practice.pdf

A.23 Chapter 4: Developer Mental Models of Public Key Cryptography: Ethics Application

Faculty of Engineering

Advertisement

Dear Sir/Madam,

My name is Nikhil Patnaik. I am a PhD student at the University of Bristol.

The reason I am contacting you is to ask if you would like to participant in a user study I am conducting as part of my research.

My research revolves around the development of usable abstractions for secure programming.

The aim of this user study is to understand your thought process as you take on the task of designing a secure encrypted connection between 2 people. This task does not require you to code, instead we ask that you think aloud while designing the system and draw to illustrate your design.

There is a second part of the study where we show you 2 implementations of a secure encrypted connection between 2 people. One implementation is written using the command line API of the OpenSSL cryptographic library, another is written using Libsodium cryptographic library.

Both OpenSSL and Libsodium are C-based cryptographic libraries. A cryptographic library is essentially a toolbox of cryptographic algorithms and functions to help you secure your applications. This is different to a cryptographic API. An API stands for Application Programming Interface. The interface (API) by which you can access the algorithms and functions in the toolbox (library) are different between OpenSSL and Libsodium.

We ask you to attempt to comprehend and explain what the code is doing in the 2 different implementations. Finally, during the exit interview, we show a third implementation through the OpenSSL C-API. We ask the participant to share their thoughts on this third implementation as well.

The results of the study will help in the development of usable abstractions to help developers secure their applications.

The duration of the study will take 1 hour 30 minutes.

You will be given a £20 Amazon gift voucher once the experiment is complete.

Let me know if you would like to participate in this user study. Further details will be given then.

Contact me at nikhil.patnaik@bristol.ac.uk

Thanks.

Nikhil Patnaik

Version 3

A.24 Chapter 4: Developer Mental Models of Public Key Cryptography: Ethics Application

Faculty of Engineering

University of BRISTOL

Participant Information Sheet

Project title: Know Your Audience

Invitation paragraph
I would like to invite you to take part in my research project. Before you decide whether or not to participate, I would like you to understand why the research is being conducted and what it would involve for you. Please ask me questions if anything is unclear.

What is the purpose of the project?

The aim of the project is to elicit mental models from observing and analysing your thought process while you design a secure encrypted connection between Alice and Bob. You must aim to fulfil the conditions below:

- If Alice wants to send a message to Bob, Alice wants to be sure that only Bob can read the message. The message cannot be read by Eve.
- If Bob receives a message he expected from Alice, Bob needs to know that Alice did indeed write the message and that it was not written by Mallory pretending to be Alice.
- After receiving a message from Alice, Bob needs to be sure that the message has not been tampered with in transmission.

During the design process we ask you to draw and think aloud as you are designing the secure encrypted connection.

We compare the results to the correct implementation of the secure encrypted connection defined in cryptography.

The second part of the experiment is a comprehension study. We show you 2 versions of the secure encrypted connection between Alice and Bob written using the OpenSSL and Libsodium cryptographic libraries. We analyse your ability to comprehend the code. We also compare the mental models elicited from you in the first part of the experiment to the implementation shown through OpenSSL and Libsodium. The implementations are a representation of the cryptographic API designer's mental model. Essentially, we compare the mental model elicited from you to the mental model of the cryptographic API designer to see if any misalignments exist.

This experiment is being conducted as part of a PhD.

Why have I been invited to participate?

The group of participants consists of software developers and university student of cryptography who show a range of experience in software development and cryptography. We have invited you because we believe you fit within this group.

Do I have to take part?
It is up to you to decide whether you wish to participate in the project. I will describe the study and go through this information sheet with you before you participate and answer any questions you might have. If you agree to take part, I will then ask you to sign a consent form. You are free to withdraw at any time, without giving a reason.

You can withdraw your participation within 30 days after the interview.

What are the possible disadvantages and risks involved in taking part in the project?
None

What are the possible benefits of taking part?
The results of the project will help in the development of usable abstractions for secure programming. The usable abstractions will help software developers use cryptographic APIs, reducing misuse and improving security of applications.

Will my participation in this project be kept confidential?

All information which is collected about you during the course of the research will be kept strictly confidential. Your interview will be stored on an encrypted filesystem in one drive. The survey itself will be delivered through Codebunk and Sketch.io, and deleted once the study is complete.

What will happen to the results of the research project?

The research project will be part of my PhD thesis. A copy of the final study can be sent to you through email if requested. “You will not be identified as a named individual in any report/publication. The report/publication will consist of an analysis of the think aloud experiment and drawing exercise in part one, and the comprehension exercise in part two”.

Who is organising and funding the research?
University of Bristol – Faculty of Engineering

Who has reviewed the study?

University of Bristol – Faculty Research Ethics Committee
Supervisor – Professor Awais Rashid

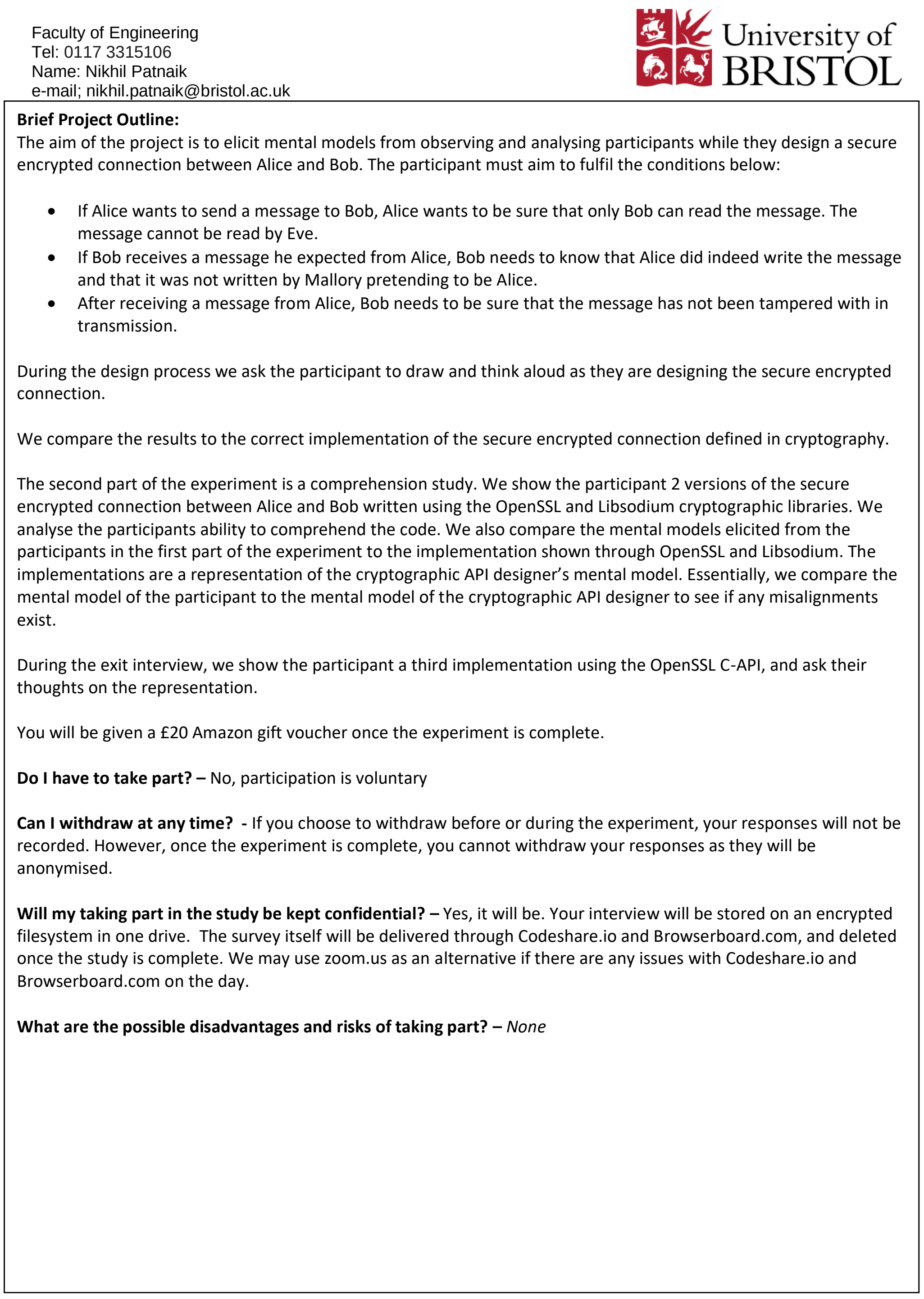
Further information and contact details
If you have further questions in relation to the study or what you are being asked to do, contact me at the email below:

Nikhil.patnaik@bristol.ac.uk

Note from Faculty of Engineering Research Ethics Committee:

If participants have any concerns related to researcher’s participation in this study, please direct them to the Faculty of Engineering Research Ethics Committee, via the Research Governance Team, research-governance@bristol.ac.uk

A.25 Chapter 4: Developer Mental Models of Public Key Cryptography: Consent Form



Faculty of Engineering
Tel: 0117 3315106
Name: Nikhil Patnaik
e-mail; nikhil.patnaik@bristol.ac.uk

University of BRISTOL

Brief Project Outline:
The aim of the project is to elicit mental models from observing and analysing participants while they design a secure encrypted connection between Alice and Bob. The participant must aim to fulfil the conditions below:

- If Alice wants to send a message to Bob, Alice wants to be sure that only Bob can read the message. The message cannot be read by Eve.
- If Bob receives a message he expected from Alice, Bob needs to know that Alice did indeed write the message and that it was not written by Mallory pretending to be Alice.
- After receiving a message from Alice, Bob needs to be sure that the message has not been tampered with in transmission.

During the design process we ask the participant to draw and think aloud as they are designing the secure encrypted connection.

We compare the results to the correct implementation of the secure encrypted connection defined in cryptography.

The second part of the experiment is a comprehension study. We show the participant 2 versions of the secure encrypted connection between Alice and Bob written using the OpenSSL and Libsodium cryptographic libraries. We analyse the participants ability to comprehend the code. We also compare the mental models elicited from the participants in the first part of the experiment to the implementation shown through OpenSSL and Libsodium. The implementations are a representation of the cryptographic API designer's mental model. Essentially, we compare the mental model of the participant to the mental model of the cryptographic API designer to see if any misalignments exist.

During the exit interview, we show the participant a third implementation using the OpenSSL C-API, and ask their thoughts on the representation.

You will be given a £20 Amazon gift voucher once the experiment is complete.

Do I have to take part? – No, participation is voluntary

Can I withdraw at any time? - If you choose to withdraw before or during the experiment, your responses will not be recorded. However, once the experiment is complete, you cannot withdraw your responses as they will be anonymised.

Will my taking part in the study be kept confidential? – Yes, it will be. Your interview will be stored on an encrypted filesystem in one drive. The survey itself will be delivered through Codeshare.io and Browserboard.com, and deleted once the study is complete. We may use zoom.us as an alternative if there are any issues with Codeshare.io and Browserboard.com on the day.

What are the possible disadvantages and risks of taking part? – *None*

Please answer the following questions to the best of your knowledge

	YES	NO
HAVE YOU:		
• been given information explaining about the study?	☐	☐
• had an opportunity to ask questions and discuss this study?	☐	☐
• received satisfactory answers to all questions you asked?	☐	☐
• received enough information about the study for you to make a decision about your participation?	☐	☐
DO YOU UNDERSTAND:		
That you are free to withdraw from the study and free to withdraw your data prior to final consent		
• at any time/ up until the point of anonymisation on xx.xx.xxxx?	☐	☐
• without having to give a reason for withdrawing?	☐	☐

I hereby fully and freely consent to my participation in this study

Participant's signature: _____________________________________ Date: ________________

Name in BLOCK Letters: _____________________________________

If you have any concerns related to your participation in this study, please direct them to the Faculty of Engineering Research Ethics Committee, via the Research Governance Team; research-governance@bristol.ac.uk

A.26 Chapter 4: Developer Mental Models of Public Key Cryptography: Initial Questionnaire

Faculty of Engineering

University of BRISTOL

Questionnaire

Project title: Know Your Audience

Initial Interview:

What is your current occupation?

[Free text]

How many years of C programming experience do you have? (Write YES in the squared parenthesis for selected option)

- < 1 year […]
- 1-2 years […]
- 3-5 years […]
- 6-10 years […]
- 11+ years […]

Rate your background/knowledge about cryptography concepts such as encryption, digests, signatures, etc. (Write YES in the squared parenthesis for selected option)

- Not knowledgeable – I do not know anything about cryptography […]
- Somewhat knowledgeable – I have a vague idea about various areas of cryptography and what they are used for […]
- Knowledgeable – I am familiar with various areas of cryptography and what they are used for […]
- Very knowledgeable – I know all/most areas of cryptography, the different available algorithms, and what they are used for […]

How often do you need to use cryptography in your software applications? (Write YES in the squared parenthesis for selected option)

- Never […]
- Rarely - I need cryptography for less than 33% of the software applications I develop […]
- Occasionally - I use cryptography in more than 33% but less than 66% of the software applications I develop […]
- Frequently - I need cryptography for more than 66% of the software applications I develop. […]

What kind of cryptography-related tasks do you usually implement in your applications?

[Free text]

Version 1

Exit Interview:

What challenges did you face while reading the OpenSSL implementation?

[Free text]

What challenges did you face while reading the Libsodium implementation?

[Free text]

Which implementation did you find to be more comprehensible and why?

[Free text]

What do you think would be a useful tool/technology/idea that can help you implement cryptography into your software applications more correctly, efficiently, etc.?

[Free text]

Do you have any additional feedback on the experiment?

[Free text]

BIBLIOGRAPHY

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, *Comparing the usability of cryptographic APIs*, in 2017 IEEE Symposium on Security and Privacy (SP), IEEE, 2017, pp. 154–171.

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

[3] D. Almeida, J. C. Campos, J. Saraiva, and J. C. Silva, *Towards a catalog of usability smells*, in Proceedings of the 30th Annual ACM Symposium on Applied Computing, ACM, 2015, pp. 175–181.

[4] D. F. Andersen and J. Rohrbaugh, *Some conceptual and technical problems in integrating models of judgment with simulation models*, IEEE Transactions on systems, Man, and Cybernetics, 22 (1992), pp. 21–34.

[5] H. Ando, R. Cousins, and C. Young, *Achieving saturation in thematic analysis: Development and refinement of a codebook*, Comprehensive Psychology, 3 (2014), pp. 03–CP.

[6] B. A. M. Andrew J. Ko, *A framework and methodology for studying the causes of software errors in programming systems*, Visual Languages and Computing, 16 (2005), pp. 41–84.

[7] A. Apvrille and M. Pourzandi, *Secure software development by example*, IEEE Security & Privacy, 3 (2005), pp. 10–17.

[8] K. Arnold, J. Gosling, D. Holmes, and D. Holmes, *The Java programming language*, vol. 2, Addison-wesley Reading, 2000.

[9] J. Arvin S. Lat, R. Xavier R. Bondoc, and K. C. V. Atienza, *Soul system: secure online usb login system*, Information Management & Computer Security, 21 (2013), pp. 102–109.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, *FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps*, in Acm Sigplan Notices, vol. 49, ACM, 2014, pp. 259–269.

[11] H. Assal and S. Chiasson, *Security in the software development lifecycle*, in Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018), 2018, pp. 281–296.

[12] J. Beaton, S. Y. Jeong, Y. Xie, J. Stylos, and B. A. Myers, *Usability challenges for enterprise service-oriented architecture APIs*, in 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE, 2008, pp. 193–196.

[13] M. Bellare and P. Rogaway, *Introduction to modern cryptography*, Ucsd Cse, 207 (2005), p. 207.

[14] Z. Benenson, F. Gassmann, and L. Reinfelder, *Android and ios users' differences concerning security and privacy*, in CHI'13 Extended Abstracts on Human Factors in Computing Systems, ACM CHI Conference on Human Factors in Computer Systems, 2013, pp. 817–822.

[15] D. J. Bernstein, T. Lange, and P. Schwabe, *The security impact of a new cryptographic library*, in International Conference on Cryptology and Information Security in Latin America, Springer, 2012, pp. 159–176.

[16] J. Bloch, *Effective Java*, Pearson Education, 2001.

[17] ——, *How to design a good API and why it matters*, in Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, ACM, 2006, pp. 506–507.

[18] E. Bodden, *Ts4j: a fluent interface for defining and computing typestate analyses*, in Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, 2014, pp. 1–6.

[19] A. Bostrom, B. Fischhoff, and M. G. Morgan, *Characterizing mental models of hazardous processes: A methodology and an application to radon*, Journal of social issues, 48 (1992), pp. 85–100.

[20] G. Boström, J. Wäyrynen, M. Bodén, K. Beznosov, and P. Kruchten, *Extending XP practices to support security requirements engineering*, in Proceedings of the 2006 international workshop on Software engineering for secure systems, ACM, 2006, pp. 11–18.

[21] V. Braun and V. Clarke, *Using thematic analysis in psychology*, Qualitative research in psychology, 3 (2006), pp. 77–101.

[22] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, *Finding and preventing bugs in JavaScript bindings*, in 2017 IEEE Symposium on Security and Privacy (SP), IEEE, 2017, pp. 559–578.

[23] C. BRUBAKER, S. JANA, B. RAY, S. KHURSHID, AND V. SHMATIKOV, *Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations*, in 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 114–129.

[24] E. BRUNSWIK, *Perception and the representative design of psychological experiments*, Univ of California Press, 1956.

[25] BSIMM.

[26] L. J. CAMP, *Mental models of privacy and security*, IEEE Technology and society magazine, 28 (2009), pp. 37–46.

[27] J. M. CARROLL AND M. B. ROSSON, *Getting around the task-artifact cycle: how to make claims and design by scenario*, ACM Transactions on Information Systems (TOIS), 10 (1992), pp. 181–212.

[28] H.-C. CHEN, H. CHANG, T.-L. KUNG, Y.-F. HUANG, Z.-M. LIN, P.-C. YEH, AND Q.-H. RUAN, *A secure color-code key exchange protocol for mobile chat application*, in Mobile Internet Security: First International Symposium, MobiSec 2016, Taichung, Taiwan, July 14-15, 2016, Revised Selected Papers 1, Springer, 2018, pp. 54–64.

[29] P. D. CHOWDHURY, J. HALLETT, N. PATNAIK, M. TAHAEI, AND A. RASHID, *Developers are neither enemies nor users: They are collaborators*, in 2021 IEEE Secure Development Conference (SecDev), IEEE, 2021, pp. 47–55.

[30] S. CLARKE AND C. BECKER, *Using the cognitive dimensions framework to evaluate the usability of a class library*, in Proceedings of the First Joint Conference of EASE PPIG (PPIG 15), 2003.

[31] J. COHEN, *A coefficient of agreement for nominal scales*, Educational and psychological measurement, 20 (1960), pp. 37–46.

[32] K. CRAIK, *The nature of explanation.,(cambridge university press: Cambridge, uk.)*, (1967).

[33] K. J. W. CRAIK, *The nature of explanation*, vol. 445, CUP Archive, 1952.

[34] J. W. CRESWELL AND J. D. CRESWELL, *Research design: Qualitative, quantitative, and mixed methods approaches*, Sage publications, 1994.

[35] K. CWALINA AND B. ABRAMS, *Framework design guidelines: conventions, idioms, and patterns for reusable .NET libraries*, Pearson Education, 2008.

[36] S. DAS, V. GOPAL, K. KING, AND A. VENKATRAMAN, *Iv= 0 security: Cryptographic misuse of libraries*, Massachusetts Institute of Technology, (2014).

[37] P. D.BERNSTEIN, T.LANGE, *NaCl.*

[38] M. DE MONTMOLLIN AND V. DE KEYSER, *Expert logic v. operator logic*, in Analysis, Design and Evaluation of Man–Machine Systems, Elsevier, 1986, pp. 43–49.

[39] U. DEKEL AND J. D. HERBSLEB, *Improving api documentation usability with knowledge pushing*, in Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, 2009, pp. 320–330.

[40] ——, *Reading the documentation of invoked API functions in program comprehension*, in 2009 IEEE 17th International Conference on Program Comprehension (ICPC 2009), IEEE, 2009, pp. 168–177.

[41] J. DES RIVIÈRES, *Eclipse APIs: Lines in the sand*, EclipseCon Retrieved March, 18 (2004), p. 2004.

[42] A. DESNOS ET AL., *Androguard*, 2011.

[43] P. T. DEVANBU AND S. STUBBLEBINE, *Software engineering for security: a roadmap*, in Proceedings of the Conference on the Future of Software Engineering, 2000, pp. 227–239.

[44] W. DIFFIE, *The first ten years of public-key cryptography*, Proceedings of the IEEE, 76 (1988), pp. 560–577.

[45] P. DOURISH, J. D. DE LA FLOR, AND M. JOSEPH, *Security as a practical problem: Some preliminary observations of everyday mental models*, in Proceedings of CHI 2003 Workshop on HCI and Security Systems, 2003.

[46] M. EGELE, D. BRUMLEY, Y. FRATANTONIO, AND C. KRUEGEL, *An empirical study of cryptographic misuse in android applications*, in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, 2013, pp. 73–84.

[47] J. H. ELLIS, *The possibility of secure non-secret digital encryption*, UK Communications Electronics Security Group, 8 (1970).

[48] ——, *The history of non-secret encryption*, Cryptologia, 23 (1999), pp. 267–273.

[49] J. H. ELLIS ET AL., *The story of non-secret encryption*, Communications–Electronics Security Group, (1987).

[50] W. ENCK, D. OCTEAU, P. D. MCDANIEL, AND S. CHAUDHURI, *A study of android application security.*, in USENIX security symposium, vol. 2, 2011.

[51] S. FAHL, M. HARBACH, T. MUDERS, L. BAUMGÄRTNER, B. FREISLEBEN, AND M. SMITH, *Why eve and mallory love android: An analysis of android ssl (in) security*, in Proceedings of the 2012 ACM conference on Computer and communications security, 2012, pp. 50–61.

[52] F.Denis, *Libsodium.*

[53] D. F. Ferraiolo, D. M. Gilbert, and N. Lynch, *An examination of federal and commercial access control policy needs*, in NIST-NCSC national computer security conference, 1995, pp. 107–116.

[54] J. W. Forrester, *Confidence in models of social behavior with emphasis on system dynamics models*, System Dynamics Group Working Paper, (1973).

[55] M. Fowler, *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 1999.

[56] M. Fowler, *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 2018.

[57] B. Friedman, D. Hurley, D. C. Howe, E. Felten, and H. Nissenbaum, *Users' conceptions of web security: a comparative study*, in CHI'02 extended abstracts on Human factors in computing systems, 2002, pp. 746–747.

[58] S. Furman, M. F. Theofanos, Y.-Y. Choong, and B. Stanton, *Basing cybersecurity training on user perceptions*, IEEE Security & Privacy, 10 (2011), pp. 40–49.

[59] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Abstraction and reuse of object-oriented design*, in European Conference on Object-Oriented Programming, Springer, 1993, pp. 406–431.

[60] P. C. Gardiner and A. Ford, *Which policy run is best, and who says so*, System Dynamics: TIMS Studies in the Management Sciences, 14 (1980), pp. 241–257.

[61] D. Garlan, R. Allen, and J. Ockerbloom, *Architectural mismatch or why it's hard to build systems out of existing parts*, in 1995 17th International Conference on Software Engineering, IEEE, 1995, pp. 179–179.

[62] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, *The most dangerous code in the world: validating ssl certificates in non-browser software*, in Proceedings of the 2012 ACM conference on Computer and communications security, 2012, pp. 38–49.

[63] M. Ghafari, P. Gadient, and O. Nierstrasz, *Security smells in android*, in 2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM), IEEE, 2017, pp. 121–130.

[64] L. Gong and G. Ellison, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, Pearson Education, 2nd ed., 2003.

[65] P. L. Gorski, L. L. Iacono, D. Wermke, C. Stransky, S. Möller, Y. Acar, and S. Fahl, *Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic API misuse*, in Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018), 2018, pp. 265–281.

[66] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*, Addison-Wesley Professional, 2000.

[67] M. Green and M. Smith, *Developers are not the enemy!: The need for usable security APIs*, IEEE Security & Privacy, 14 (2016), pp. 40–46.

[68] T. R. Green, *Cognitive dimensions of notations*, People and computers V, (1989), pp. 443–460.

[69] T. R. G. Green and M. Petre, *Usability analysis of visual programming environments: a 'cognitive dimensions' framework*, Journal of Visual Languages & Computing, 7 (1996), pp. 131–174.

[70] T. Grill, O. Polacek, and M. Tscheligi, *Methods towards API usability: a structural analysis of usability problem categories*, in International conference on human-centred software engineering, Springer, 2012, pp. 164–180.

[71] E. S. Group, *ESG Survey Report: Modern Application Development Security*.

[72] P. Gutmann, *Peter gutmann. cryptlib security toolkit: user's guide and manual*, Gutmann, Peter, (1995).

[73] ——, *The design of a cryptographic security architecture.*, in USENIX Security Symposium, 1999.

[74] ——, *Lessons learned in implementing and deploying crypto software.*, in Usenix Security Symposium, 2002, pp. 315–325.

[75] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, *Security requirements engineering: A framework for representation and analysis*, IEEE Transactions on Software Engineering, 34 (2008), pp. 133–153.

[76] J. Hallett, N. Patnaik, B. Shreeve, and A. Rashid, *"do this! do that!, and nothing will happen" do specifications lead to securely stored passwords?*, in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 486–498.

[77] P. Harms and J. Grabowski, *Usage-based automatic detection of usability smells*, in International Conference on Human-Centered Software Engineering, Springer, 2014, pp. 217–234.

[78] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, *Mining your ps and qs: Detection of widespread weak keys in network devices*, in Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), 2012, pp. 205–220.

[79] M. Henning, *API design matters*, Queue, 5 (2007), pp. 24–36.

[80] D. Hoffman, *On criteria for module interfaces*, IEEE transactions on Software Engineering, 16 (1990), pp. 537–542.

[81] R. Holcomb and A. L. Tharp, *An amalgamated model of software usability*, in Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, IEEE, 1989, pp. 559–566.

[82] L. L. Iacono and P. L. Gorski, *I do and i understand. not yet true for security apis. so sad*, in European Workshop on Usable Security, 2017.

[83] I. International Conference on Software Engineering, *SLR: From Saltzer & Schroeder to 2021...*

[84] M. Jacques, *API usability: guidelines to improve your code ease of use*, 2004.
Code Project.

[85] P.-H. Kamp, *Please put OpenSSL out of its misery.*, ACM Queue, 12 (2014), pp. 20–23.

[86] T. G. Kannampallil and J. M. Daughtry III, *Handling objects: a scenario based approach*, in Proceedings of the 24th annual ACM international conference on Design of communication, ACM, 2006, pp. 92–98.

[87] M. Kauer, S. Günther, D. Storck, and M. Volkamer, *A comparison of american and german folk models of home computer security*, in International conference on human aspects of information security, privacy, and trust, Springer, 2013, pp. 100–109.

[88] A. J. Ko and B. A. Myers, *A framework and methodology for studying the causes of software errors in programming systems*, Journal of Visual Languages & Computing, 16 (2005), pp. 41–84.

[89] A. J. Ko, B. A. Myers, and H. H. Aung, *Six learning barriers in end-user programming systems*, in 2004 IEEE Symposium on Visual Languages-Human Centric Computing, IEEE, 2004, pp. 199–206.

[90] A. J. Ko and Y. Riche, *The role of conceptual knowledge in API usability*, in Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium, IEEE, 2011, pp. 173–176.

[91] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, et al., *Cognicrypt: Supporting developers in using cryptography*, in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 931–936.

[92] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, *Crysl: An extensible approach to validating the correct usage of cryptographic apis*, IEEE Transactions on Software Engineering, (2019).

[93] B. T. Laboratory, *Final Report on Project C43*, 1944.

[94] J. R. Landis and G. G. Koch, *An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers*, Biometrics, 33 (1977), pp. 363–374.

[95] S. Lipner, *The trustworthy computing security development lifecycle*, in 20th Annual Computer Security Applications Conference, IEEE, 2004, pp. 2–13.

[96] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, *Building more usable APIs*, IEEE software, 15 (1998), pp. 78–86.

[97] N. R. Mead and T. Stehney, *Security quality requirements engineering (SQUARE) methodology*, vol. 30, ACM, 2005.

[98] J. Meier, *Web application security engineering*, IEEE Security & Privacy, 4 (2006), pp. 16–24.

[99] A. Mendoza and G. Gu, *Mobile application web API reconnaissance: Web-to-mobile inconsistencies & vulnerabilities*, in 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 756–769.

[100] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, *Secure coding practices in java: Challenges and vulnerabilities*, in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 372–383.

[101] R. C. Merkle, *Secure communications over insecure channels*, Communications of the ACM, 21 (1978), pp. 294–299.

[102] Microsoft, *Security Development Lifecycle*.

[103] G. A. Miller, G. Eugene, and K. H. Pribram, *Plans and the structure of behaviour*, in Systems Research for Behavioral Sciencesystems Research, Routledge, 2017, pp. 369–382.

[104] J. Mills, A. Bonner, and K. Francis, *The development of constructivist grounded theory*, International journal of qualitative methods, 5 (2006), pp. 25–35.

[105] K. Mindermann, P. Keck, and S. Wagner, *How usable are rust cryptography APIs?*, in 2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018, 2018, pp. 143–154.

[106] R. Molich and J. Nielsen, *Improving a human-computer dialogue*, Communications of the ACM, 33 (1990), pp. 338–348.

[107] I. Muslukhov, Y. Boshmaf, and K. Beznosov, *Source attribution of cryptographic api misuse in android applications*, in Proceedings of the 2018 on Asia Conference on Computer and Communications Security, 2018, pp. 133–146.

[108] B. A. Myers and J. Stylos, *Improving API usability*, Communications of the ACM, 59 (2016), pp. 62–69.

[109] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, *Jumping through hoops: Why do Java developers struggle with cryptography APIs?*, in Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 935–946.

[110] NCSC, *Data breach of 500m Yahoo accounts*.

[111] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, *A stitch in time: Supporting android developers in writingsecure code*, in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1065–1077.

[112] J. Nielsen, *Enhancing the explanatory power of usability heuristics*, in Proceedings of the SIGCHI conference on Human Factors in Computing Systems, ACM, 1994, pp. 152–158.

[113] ——, *Usability metrics: Tracking interface improvements*, IEEE Software, 13 (1996), pp. 1–2.

[114] J. Nielsen and R. Molich, *Heuristic evaluation of user interfaces*, in Proceedings of the SIGCHI conference on Human factors in computing systems, ACM, 1990, pp. 249–256.

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

[116] N.Patnaik, J.Hallett, M.Tahaei, and A.Rashid, *If you build it, will they come? developer recruitment for security studies*, 2022.

[117] B. of Apps, *App Store Data (2023)*.

[118] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, and Y. Brun, *{API} blindspots: Why experienced developers write vulnerable code*, in Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018), 2018, pp. 315–328.

[119] M. O'Neill, S. Heidbrink, S. Ruoti, J. Whitehead, D. Bunker, L. Dickinson, T. Hendershot, J. Reynolds, K. Seamons, and D. Zappala, *Trustbase: an architecture to repair and strengthen certificate-based authentication*, in 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 609–624.

[120] M. O'Neill, S. Heidbrink, J. Whitehead, T. Perdue, L. Dickinson, T. Collett, N. Bonner, K. Seamons, and D. Zappala, *The secure socket {API}:{TLS} as an operating system service*, in 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 799–816.

[121] OWASP, *OWASP Developer Guide*.

[122] ——, *OWASP SAMM Project*.

[123] J. F. Pane and B. A. Myers, *Usability issues in the design of novice programming systems*, tech. rep., Carnegie-Mellon University, 1996.

[124] C. Parnin, C. Treude, L. Grammel, and M. A. Storey, *Crowd documentation: Exploring the coverage and the dynamics of API discussions on stack overflow*, Georgia Institute of Technology, Tech. Rep, (2012).

[125] N. Patnaik, A. C. Dwyer, J. Hallett, and A. Rashid, *Slr: From saltzer & schroeder to 2021...*, ACM Transactions on Software Engineering and Methodology, (2022).

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

[127] S. J. Payne, *A descriptive study of mental models*, Behaviour & Information Technology, 10 (1991), pp. 3–21.

[128] P.Gutmann, *Cryptlib*.

[129] M. Piccioni, C. A. Furia, and B. Meyer, *An empirical study of API usability*, in 2013 ACM/IEEE international symposium on Empirical Software Engineering and Measurement, IEEE, 2013, pp. 5–14.

[130] P. G. Polson and C. H. Lewis, *Theory-based design for easily learned interfaces*, Human-computer interaction, 5 (1990), pp. 191–220.

[131] A. Poppele, R. Eichler, and R. Jäger, *Classification of cryptographic libraries*, (2017).

[132] G. P. Richardson, D. F. Andersen, T. A. Maxwell, and T. R. Stewart, *Foundations of mental model research*, in Proceedings of the 1994 International System Dynamics Conference, EF Wolstenholme, 1994, pp. 181–192.

[133] C. RIVAS, *Coding and analysing qualitative data*, Researching society and culture, 3 (2012), pp. 367–392.

[134] M. P. ROBILLARD, *What makes APIs hard to learn? Answers from developers*, IEEE software, 26 (2009), pp. 27–34.

[135] M. P. ROBILLARD AND R. DELINE, *A field study of api learning obstacles*, Empirical Software Engineering, 16 (2011), pp. 703–732.

[136] J. ROHRBAUGH AND D. ANDERSEN, *Specifying dynamic objective functions: Problems and possibilities*, Dy-namiea, 9 (1983), pp. 1–7.

[137] A. SACHITANO, R. O. CHAPMAN, AND J. HAMILTON, *Security in software architecture: a case study*, in Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004., IEEE, 2004, pp. 370–376.

[138] J. SALDAÑA, *The coding manual for qualitative researchers*, Sage, 2015.

[139] J. H. SALTZER, *Protection and the control of information sharing in Multics*, Communications of the ACM, 17 (1974), pp. 388–402.

[140] J. H. SALTZER AND M. D. SCHROEDER, *The protection of information in computer systems*, Proceedings of the IEEE, 63 (1975), pp. 1278–1308.

[141] S. SARKAR, G. M. RAMA, AND A. C. KAK, *API-based and information-theoretic metrics for measuring the quality of software modularization*, IEEE Transactions on Software Engineering, 33 (2006), pp. 14–32.

[142] T. SCHELLER AND E. KÜHN, *Automated measurement of api usability: The api concepts framework*, Information and Software Technology, 61 (2015), pp. 145–162.

[143] F. B. SCHNEIDER, *Enforceable security policies*, tech. rep., Cornell University, 1999.

[144] R. SEACORD, *Top 10 secure coding practices*.

[145] SECURE SOFTWARE, INC, *The CLASP Application Security Process*, 2005.

[146] S. SHUAI, D. GUOWEI, G. TAO, Y. TIANCHANG, AND S. CHENJIE, *Modelling analysis and auto-detection of cryptographic misuse in android applications*, in 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, IEEE, 2014, pp. 75–80.

[147] G. SINDRE AND A. L. OPDAHL, *Eliciting security requirements with misuse cases*, Requirements engineering, 10 (2005), pp. 34–44.

[148] M. T. SIPONEN, *Critical analysis of different approaches to minimizing user-related faults in information systems security: implications for research and practice*, Information Management & Computer Security, 8 (2000), pp. 197–209.

[149] D. C. SMITH, C. IRBY, R. KIMBALL, AND B. VERPLANK, *Designing the Star User Interface*, BYTE, (1982), pp. 242–282.

[150] R. E. SMITH, *A contemporary look at Saltzer and Schroeder's 1975 design principles*, IEEE Security & Privacy, 10 (2012), pp. 20–25.

[151] STATISTA, *Number of apps available in leading app stores.*

[152] ——, *Number of available applications in the Google Play Store from December 2009 to March 2023.*

[153] ——, *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030.*

[154] C. STRANSKY, Y. ACAR, D. C. NGUYEN, D. WERMKE, D. KIM, E. M. REDMILES, M. BACKES, S. GARFINKEL, M. L. MAZUREK, AND S. FAHL, *Lessons learned from using an online platform to conduct large-scale, online controlled security experiments with software developers*, in 10th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 17), 2017.

[155] A. STRAUSS AND J. M. CORBIN, *Grounded theory in practice*, Sage, 1997.

[156] J. STYLOS AND B. MYERS, *Mapping the space of API design decisions*, in IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007), IEEE, 2007, pp. 50–60.

[157] SYNK.IO, *Secure Software Development Lifecycle (SSDLC).*

[158] TECHTARGET, *MD5.*

[159] T. N. Y. TIMES, *RSA Faces Angry Users After Breach.*

[160] I. A. TONDEL, M. G. JAATUN, AND P. H. MELAND, *Security requirements for the rest of us: A survey*, IEEE software, 25 (2008), pp. 20–27.

[161] C. TREUDE AND M. P. ROBILLARD, *Augmenting API documentation with insights from stack overflow*, in Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, IEEE, 2016, pp. 392–403.

[162] M. UKROP AND V. MATYAS, *Why Johnny the developer can't work with public key certificates*, in Cryptographers' Track at the RSA Conference, Springer, 2018, pp. 45–64.

[163] M. Volkamer and K. Renaud, *Mental models–general introduction and review of their application to human-centred security*, in Number Theory and Cryptography, Springer, 2013, pp. 255–280.

[164] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, *Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it*, in Proceedings of the 29 th USENIX Security Symposium (USENIX) Security, vol. 20, 2020.

[165] R. Wash, *Folk models of home computer security*, in Proceedings of the Sixth Symposium on Usable Privacy and Security, 2010, pp. 1–16.

[166] D. Weirich and M. A. Sasse, *Pretty good persuasion: a first step towards effective password security in the real world*, in Proceedings of the 2001 workshop on New security paradigms, 2001, pp. 137–143.

[167] C. Wohlin, *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, New York, NY, USA, 2014, Association for Computing Machinery.

[168] J. Wu and D. Zappala, *When is a tree really a truck? exploring mental models of encryption*, in Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018), 2018, pp. 395–409.

[169] M. Zibran, *What makes APIs difficult to use*, International Journal of Computer Science and Network Security, 8 (2008), pp. 255–261.

[170] M. F. Zibran, F. Z. Eishita, and C. K. Roy, *Useful, but usable? factors affecting the usability of APIs*, in 2011 18th Working Conference on Reverse Engineering, IEEE, 2011, pp. 151–155.