uOttawa

GhostBuy:
An *All-Steps Anonymous Purchase Platform (ASAPP)* based on *Separation of Data*

by

Fabian Willems

A thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Master of Science
in
Electronic Business Technologies (EBT)

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Preface

The research presented in this thesis is entirely our own work. Sources are declared throughout the thesis and – regarding adapted / adopted code snippets – in the source code of the prototype implementation that we provide with the thesis.

However, in Appendix V we provide an overview of sources that were particularly useful for the prototype implementation. We hope that others will find them as useful as we did.

Abstract

In recent years – and especially since the beginning of the COVID-19 pandemic – online shopping has become a part of everyday life for many people. Yet, in contrast to buying at a traditional retail store, staying anonymous is at least difficult if not impossible when shopping online – in particular, when physical goods are to be delivered. From the customer perspective, reasons for seeking anonymity when shopping online can be manifold, for example some do not want anyone to know about their purchases, others do not want their data to be used by Big Data-enabled online retailers. From the point of view of online retailers, the prospect of anonymous online shopping should therefore not only be seen as a threat to their data-driven business models, but also as an opportunity to attract new customers.

In this thesis we search and find support in the literature regarding the question whether there is indeed a demand for anonymous online shopping, and we discuss system architecture designs that were proposed by other authors for potentially realizing what we call *All-Steps Anonymous Purchase Platforms (ASAPP)*. We propose a new architecture design that improves earlier work by realizing the concept of *Separation of Data* within a single platform: *GhostBuy*.

We implement a working prototype of this platform that demonstrates not only the fundamental feasibility of the architecture but also that such a platform can be realized with a look-and-feel similar to that of common online shops. We also propose solutions for certain related aspects that are particularly important in the context of such a platform, as for example a guaranteed use of secure user passwords or application-level database encryption.

We evaluate to what extent the proposed architecture and prototype preserve the customers' anonymity/privacy, showing that the prototype provides it to the maximum possible extent that can be achieved based on the proposed architecture. We also show that the system provides 256-bit security against all but one considered cryptographic and mis-authentication attack vectors and discuss how this can also be achieved for the remaining attack vector. Closing our evaluation, we show how well the platform could presumably be deployed in the real world. Finally, limitations, possible improvements, and potential further future work are discussed and proposed.

Acknowledgements

I would like to thank my wife for her continuous support before and during my studies in the University of Ottawa's EBT program. Without you, I would not have been able to finish the program, and, in fact, I never would have started it. I also want to thank my kids for being understanding or at least accepting when I was not at home or just not available due to taking classes or working on assignments and this thesis. I am looking forward to a time of just being a dad – and a husband!

Also, I like to give a big thank you to my brother Sebastian for doing such a great job of implementing my rough logo-design idea – it looks fantastic!

Being allowed to enter the EBT program would not have been possible without the support of my current and former supervisors Monika and Andreas and my former professors at the *Fachhochschule der Wirtschaft (FHDW)*, Dr. Künzel and Dr. Baeumle-Courth. Thank you for your guidance and support, your lectures, and the reference letters for my application for the EBT program.

A big thank you also goes to Ms. Carroll-Scott and Ms. Schwabe from the *FHDW's International Office*: Your spontaneous willingness to translate my study documents was invaluable!

I would also like to acknowledge the directors and lecturers of the EBT (now DTI) program at the University of Ottawa. The program has provided me with a wealth of valuable knowledge and skills which, not least, have enabled me to write this thesis.

Finally, I would like to thank my supervisor Professor Carlisle Adams, firstly, for certainly significantly improving my prospects of successfully applying for the EBT program through his willingness to be my supervisor. Secondly, I want to thank him for his continuous guidance and support throughout my studies: I absolutely enjoyed working and talking with you and very much hope that we will stay in contact.

Table of Contents

List of Figures

List of Tables

List of Abbreviations & Technical Terms

AES ..... *Advanced Encryption Standard*
AES256 ..... *AES with 256-bit key*
AES-GCM ..... *AES in Galois/Counter Mode*
AJAX ..... *Asynchronous JavaScript and XML*
API ..... *Application Programming Interface*
ASAPP ..... *All-Steps Anonymous Purchase Platform*
BLOB ..... *Binary Large OBject*
CONF ..... *Conference Proceedings*
COVID-19 ..... *Coronavirus disease 2019*
CSRNG ..... *Cryptographically Secure Random Number Generator*
CSS ..... *Cascaded Style Sheets*
DAO ..... *Data Access Object*
DB ..... *Database*
DBO ..... *DataBase Objects*
DOM ..... *Document Object Model*
DTO ..... *Data Transfer Object*
ECommerce ..... *Electronic Commerce*
GUI ..... *Graphical User Interface*
HTML ..... *Hyper Text Markup Language*
IDE ..... *Integrated Development Environment*
IndexedDB ..... *Indexed Database API, web browser API for NoSQL database*
IP address ..... *Internet Protocol address*
JOUR ..... *Journal Article*
JSON ..... *JavaScript Object Notation*
JSP ..... *Jakarta Server Pages (previously Java Server Pages)*
NIST ..... *National Institute of Standards and Technology (US)*
OAEP ..... *Optimal Asymmetric Encryption Padding*
ORM ..... *Object-Relational Mapping*
OT ..... *Oblivious Transfer*
PAT ..... *Patent*
PBKDF2 ..... *Password-Based Key Derivation Function 2*
PPEP ..... *Privacy Preserving E-Commerce Protocol*
QR code ..... *Quick Response code*
SHA ..... *Secure Hash Algorithms, a family of cryptographic hash functions*
SHA-2 ..... *Secure Hash Algorithm 2*
SHA-256 ..... *SHA-2 with 256 bits output*
SHA-512 ..... *SHA-2 with 512 bits output*
THES ..... *Thesis*
TLS ..... *Transport Layer Security*
URL ..... *Uniform Resource Locator*
USD ..... *United States (US) Dollar*

1. Introduction

How to stay anonymous while purchasing goods? This can be easily done in a traditional retail store by paying cash and staying mute about oneself, e.g. by not talking to the cashier nor using a rewards card. Nevertheless, the customer will in most cases be seen by staff, e.g. the cashier, or even be recognized by staff members or other customers. In order to avoid even these interactions, a customer for example can use vending machines, but, evidently, these do not provide the same range of products as retail stores and might not accept cash in all cases.

In contrast, online retailers carry millions of different products and to perform a purchase, generally, no personal interaction is necessary [1]. Accordingly, shopping online seems to be an alternative for customers seeking to stay anonymous during their purchase. But in fact, standard processes of shopping online require giving away personal information for example for payment and delivery. Moreover, anonymous online shopping of physical goods is nowadays in reality limited to using a patchwork of technologies and services such as the Tor browser for browsing and ordering goods, such as gift cards, prepaid debit cards, third-party masking services (*escrow services)* as well as certain types of mobile wallets and digital currencies for payment and finally mail forwarding services for delivery [2], [3]. These technologies and measures have the disadvantage of not being very convenient to use and/or they are not completely anonymous, e.g. seeing that purchases can be linked to the customer by third parties (e.g. mail forwarding or escrow service provider).

Therefore, the research presented in this thesis sought to solve the problem of how to provide the possibility of anonymously purchasing goods online, without the necessity to use several tools and services but instead covering all steps of the online purchase (browsing/ordering, payment, delivery) within one platform – that we accordingly name an *All-Steps Anonymous Purchase Platform (ASAPP)*. Additionally, the research aimed at designing this platform in such a way that (to a realistic extent and under *normal circumstances*) it is ensured that purchases cannot be linked to the customer either by the parties involved (bank, online shop, shipper) or by the platform operators themselves.

1.1. Motivation

Although it seems plausible that there is a need for such a platform, we would like to summarize our results presented in an earlier literature review [3] (see Appendix W) which searched previous literature for support of such a need. Section 1.1.1 will define related concepts and terms and Section 1.1.2 will summarize first the support we found in the literature for a demand towards anonymous online shopping and second our findings regarding the platform designs that were proposed by other authors.

1.1.1. Terms and Definitions

To create a common understanding of the problem addressed in this thesis the following describes the process of shopping online as understood in this context. Subsequently, relevant terms are named and defined:

1. A consumer opens an online shop's website and browses or searches its catalog. Then, the consumer selects the products that shall be purchased, typically by adding them to a virtual shopping cart.

2. To place the order, the consumer specifies payment information (e.g. credit card or PayPal details). In the case that physical products (as opposed to electronic products such as a music file download) must be delivered the consumer also specifies a shipping address.

3. The payment information is checked by the online shop (including transmittal of data to and from third parties such as the consumer's financial institution). If necessary, the consumer will authorize the payment with the financial institution.

4. The purchased physical products are delivered to the consumer by a shipper/courier.

5. Optionally, it can happen that the consumer needs to or wants to send products back to the online shop, e.g. if they do not fit or appeal, if they are damaged or if they break at a later point of time (warranty handling). To do so, the consumer sends the products to be returned back to the online shop, often enclosing a return form and/or indicating the products to be returned on the online shop's website. Once the online shop has

received the products, it will process them accordingly, e.g. by refunding the consumer or by shipping repaired products or replacements.

Figure 1 shows the simplified data flow of the non-anonymous online shopping process excluding the optional returning of products (the numbering does not fully correspond to the above process steps).

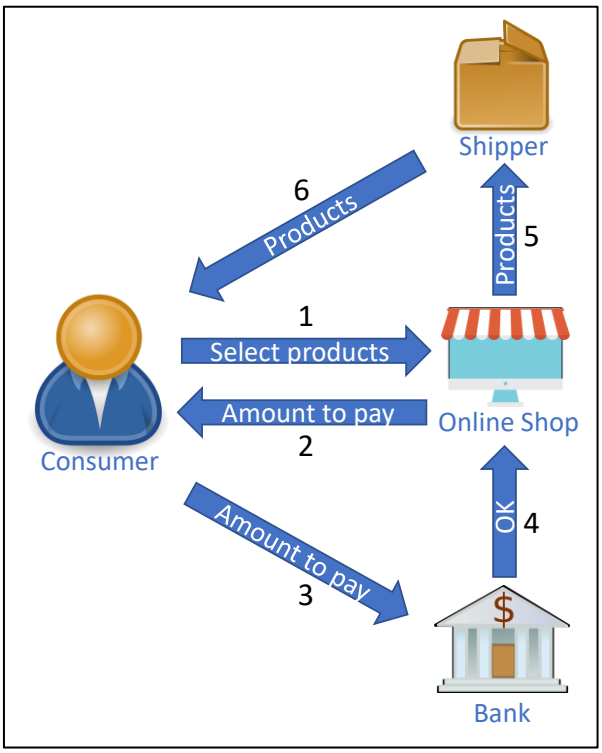

Figure 1: Simplified non-anonymous online shopping process data flow

The interactions between the parties involved (consumer, online shop, shipper, financial institutions) in most cases are *transactions of information* (including electronic money transfers). If these transactions cannot be linked to a certain person (the consumer) these transactions are called *anonymous transactions*. If these transactions can be linked to a certain person but only via matching it with *additional data* they are called *pseudonymous transactions* [4, p. 85]. Consequently, we argue that also transactions other than *transactions of information*, e.g. the delivery of products, can equally be described as *pseudonymous* or *anonymous,* depending on the possibility of linking them to a certain person with or without matching it with additional data. Similarly, Groppe et al. [2] call the whole process of online shopping *anonymous shopping* when it cannot be linked to the consumer by any party involved (other than the consumer) and additional data – such as the shopping history – does not make it possible to create a link, either. It should be noted that, in this case, this characteristic only applies to one involved party acting on its own. If several parties collaborate in a malicious way, it would be possible to create a link.

However, due to practical and legal reasons, real online shops and/or cooperation partners that comply with legal requirements must always have a way to link transactions to the parties involved. Examples for these reasons are the handling of returns and the avoidance of fraud, respectively to allow for criminal investigations [2], [4]. It also has to be noted that in order to

process payments and deliveries it must be possible, for example, to link orders to the respective payments and products to the addresses they shall be shipped to, but – to ensure anonymity – that no party in the process can make the link on its own, respectively can get more information than it needs in order to fulfill its duties.

Regarding the distinction between *anonymous* and *pseudonymous*, our literature review showed that not all researchers distinguish these terms, but for various reasons use the term *anonymous* even in the case that only *pseudonymous* properties are given [3, p. 5]. Throughout this document we also use the term *anonymous*, hereby meaning that under *normal circumstances* a *link* between transactions – here purchases – and the consumers involved cannot be made but that it is possible where necessary (more details in Section 6.2). "Necessary" for example in this case means that a criminal investigation requires to link a certain purchase to a certain consumer. Additionally, *normal circumstances* also refer to that a computationally bounded attacker will not be able to break the system, i.e. will not be able to link the consumer to the purchase.

We argue that from the consumer's perspective *staying anonymous under normal circumstances,* especially not being linked to a certain purchase (of certain products), is the required property and the maximum possible approximation to actual anonymity with regards to the legal and practical reasons mentioned before.

Since in the next section we will refer to privacy-related research we additionally want to define the term *privacy* and set it into context:

As shown above, it is apparent that in order to purchase products online the consumer is required to disclose private information, e.g. to receive physical products or to perform the payment. At first glance this is at odds with *privacy* which according to Belanger et al. is "the ability to manage information about oneself" [5, p. 249] in the sense that in this case the consumer cannot choose not to disclose any private information. Yet, if the consumer must disclose private information when shopping online, an *anonymous* approach – in the meaning as described above – allows the consumer to maintain privacy. This shows that *anonymity* in online shopping is a technical measure to ensure *privacy*, a relationship that in recent years has sometimes been accordingly expressed as "*privacy-preserving"* [6], [7].

1.1.2. Research Gap

Our literature review did not find any research which – based on above definitions – explicitly examines if there is a need for anonymous online shopping [3, p. 5]. However, as we show in more detail in Chapter 2, we have found extensive literature showing that consumers are concerned about maintaining their privacy when shopping online and that this has an impact on their shopping intentions. The ability to shop anonymously online would therefore be desirable, at least from the consumer's point of view.

Accordingly, there are also numerous proposals from other authors on how anonymous online shopping could be realized. However, as our review shows, these proposals cannot be implemented as real platforms for a variety of reasons – at least not in a timely manner. This thesis aims to close this research gap by modifying, extending, and detailing a certain one of the reviewed designs.

A more detailed discussion of the above findings can be found in Chapter 2.

1.2. Thesis Contributions

In this thesis we design, implement, and evaluate the prototype of an *All-Steps Anonymous Purchase Platform* that allows for anonymous shopping at online retailers. To achieve this:

- We summarize our earlier results of our literature research regarding the demand for anonymous online shopping and the demand for an *All-Steps Anonymous Purchase Platform*. Additionally, we review if new research on the topic has been performed, recently.
- We modify the technical design by Groppe et al. [2] by condensing the therein proposed concept of *Separation of Data* into a single platform.
- We detail the architecture and dataflows that are necessary to implement a working prototype of this platform.
- We implement a prototype of the platform in a mostly generic way, i.e. that in general it could be integrated with various online retailers. For evaluation/demonstration purposes, we also implement the necessary connector to integrate it with several marketplaces of the eBay e-commerce platform. The prototype hereby includes functional, albeit imperfect solutions to several problems which were not crucial to the

implementation of the platform's basic approach to anonymity. We nevertheless considered these as we were looking to build and demonstrate a prototype of the platform's complete ecosystem. These problems respectively solutions include checking search terms for private information, removing external sources from embedded websites, generating secure passwords, and encrypting database information.

- We outline the organization of the workflow – specifically regarding the handling of physical products when shipping them to a customer – that could be established within the platform. Also, we implement the rudimentary user interface and its backing functionality that could be used in the context of this workflow.
- We evaluate the prototype against the criteria *Privacy & Anonymity*, *Security* and *Deployability* with each consisting of several properties.
- We outline a possible way to implement an interface that would allow for law enforcement agencies to obtain required information without leaking other information to the agency and at the same time hiding from the platform which information the agency obtained.

1.3. Thesis Methodology and Organization

The research presented in this thesis followed the *Design Science Research Methodology (DSRM)* whose goal is to design and develop an "*artifact"*. Consequently, this thesis covers the different *Design Science Research (DSR)* process steps as shown in Figure 2 and the thesis' organization also follows their sequence. Section 1.1 and – in more detail – Chapter 2 define the problem and show its importance in order to motivate the conducted research: We identify a research gap by reviewing existing literature on privacy in the context of online shopping as well as proposed designs that shall enable consumers to anonymously shop online. Chapters 3 to 6 relate to the iterated part of the design process: Chapter 3 discusses the *Objectives* of the research which are first to extend earlier work in order to design an architecture for an *All-Steps Anonymous Purchase Platform* and second to implement a prototype of the platform in order to evaluate it. Chapter 4 describes the general research approach by detailing the identified main research problems, the search of the related literature, certain important decisions regarding the implementation and the subsequent iterated design and

implementation process. The results of this process are demonstrated in the form of the final artifact, i.e. the implemented prototype, whose internal workings are explained in Chapter 5. A detailed evaluation of the architecture and prototype – including the definition of the to be evaluated criteria – is performed in Chapter 6. This thesis and the accompanying materials (source code, documentation) themselves cover the last DSR process step *Communication* (*of results*).

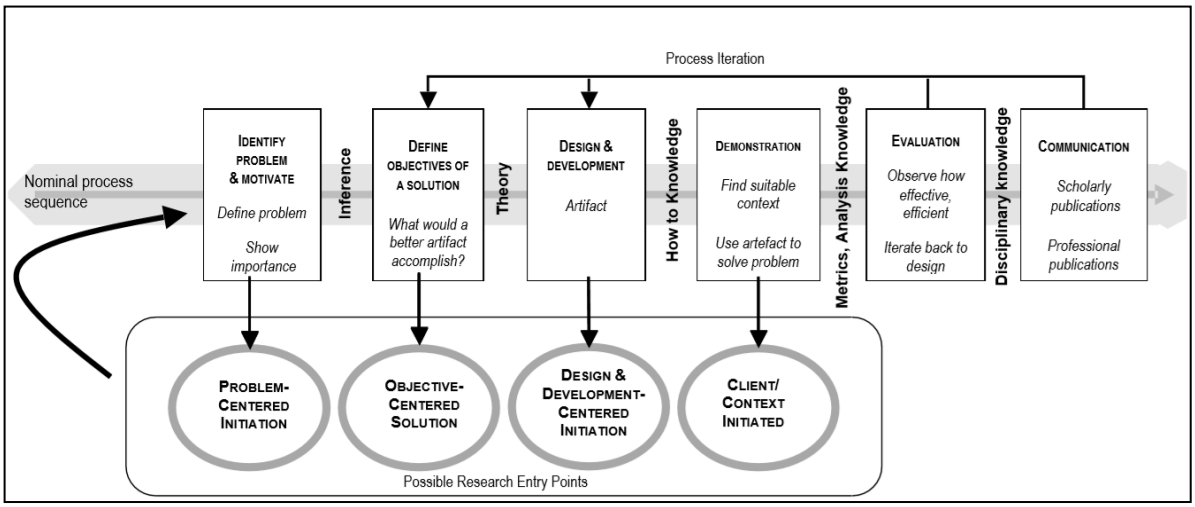

Figure 2: Design Science Research Methodology (DSRM) process model [8, Fig. 1]

Iterating these process steps allowed for reaching the best solution to be found within the limits of this research. De facto, though, only the process steps *Design & Development*, *Demonstration, Evaluation* and *Communication* haven been iterated, meaning that we did not revise the overall objectives described in Chapter 3. Nevertheless, this research's extent, especially regarding the sophistication level of certain parts of the *Evaluation* (implementation/simulation) had to be revised during the process.

We conclude in Chapter 7 and outline potential future work.

In preparation of the chapters covering the iterated DSR process steps, the following Section 1.4 clarifies the use of certain synonyms as well as related terms.

1.4. Gendering, Synonyms & Related Terms

To facilitate the reading flow, we do not use gender-specific personal pronouns but instead use the pronoun "they"[1] and its inflected forms "them", "their", "themselves" etc. as proposed and discussed in earlier and recent work [9]–[11] and as it is recommended for use in written language, including formal language, legal language and academic writing [12]–[15].

To avoid confusion regarding the use of synonymous terms, we have generally used only one term for one thing. However, some terms – though being very similar – have slightly different meanings, also depending on context, and it would be imprecise to not use the correct terms. Also, other authors have used their preferred terms, which we had to adopt in some cases. Other terms are related but have very different meanings.

We provide an overview of important synonyms and related terms used in this thesis in Table 1.

Table 1: Synonyms and Related Terms

Terms	Explanation/Relation
Consumer, Customer, User	Consumers are the indefinite number of people who use online shopping. Only the consumers using a certain service – e.g. our proposed platform – are called customers (of this service). The technical term we use for a customer who uses the platform is *the user*, e.g. as in *user* or *user account*.
Customer Device (Client), Client Application, Browser, JavaScript, HTML, CSS, Web Crypto API	A customer has a device (e.g. PC) on which they use a browser, which runs the *Client Application* by rendering HTML/CSS and executing JavaScript that also invokes the *Web Crypto API*. Since the *Client Application* is provided by the *Portal Application* it is also rarely referred to as the *Portal Application's* client-side functionality.
(User) Authentication, Sign-Up, Sign-In	A user who has to / wants to authenticate themselves must sign up or sign in. Servers as well as web service & database clients authenticate themselves, too, but e.g. through certificates or predefined passwords.
GhostBuy, Platform, Architecture, Prototype	In this thesis we propose the architecture of an *All-Steps Anonymous Purchase Platform* that we call *GhostBuy* and which we also implement as a prototype.
E-Commerce Website, (Online) Shop/Retailer, (eBay) Marketplaces	A consumer can order goods from an e-commerce website, which can be a simple online shop / online retailer or an e-commerce marketplace, such as eBay. Our proposed platform could in general integrate with any e-commerce

[1] So-called *Singular "They"* [9]–[13], [15]

Terms	Explanation/Relation
	website, but the prototype only implements this integration for multiple eBay marketplaces which we simply call “Shop” in the GUI.
Shopping Cart (Collection)	Common online shops manage a virtual shopping cart for their customers. In the GUI of our prototype we use this term as well, since that is what customers are used to. However, in fact our prototype does manage separate shopping carts for the respective shops, thus technically being a collection of shopping carts that we simply call *Shopping Cart Collection*.
Java Server Pages, JavaServer Pages, Jakarta Server Pages	Our web server implementation uses Java Server Pages / JavaServer Pages – JSP for short – which have recently been renamed to “Jakarta Server Pages”. Throughout this thesis we use the new term “Jakarta Server Pages”.

2. Related Work

In this chapter we give a more detailed summary of our earlier literature review [3] that motivated this thesis as briefly shown in Section 1.1. Section 2.1 will show the support we found in the literature for a demand towards anonymous online shopping and Section 2.2 covers the reviewed architectures that potentially enable consumers to anonymously shop online. In this section we also examine why these designs presumably have not (yet) been implemented in the real world, respectively why this will not happen in the near future. At last we outline how we want to improve the design by Groppe et al. [2] to close the identified research gap.

We close Chapter 2 with Section 2.3 in which we revisit the literature to examine the recent research regarding privacy and anonymity in online shopping as well as proposals for according designs of anonymous e-commerce platforms.

2.1. Demand for Anonymous Online Shopping

In our literature review [3] we did not find any research which – based on the definitions given in Section 1.1.1– explicitly examines if there is a need for anonymous online shopping [3, p. 5].

Nevertheless, our review shows that there is extensive research on *privacy* in the sense of *privacy as perceived by consumers shopping online* [16, p. 568], [17, p. 135] and *privacy concerns* [17]–[20] where *privacy concerns* express concerns regarding "the ability to manage information about oneself" [5, p. 249]. Overall, the reviewed research focuses on the direct and indirect influence of *perceived privacy* and *privacy concerns* on the consumer's *attitude to e-shopping* and more specifically to the *consumer's trust in the online retailer* and the consumer's online shopping *purchase intentions* [5], [16]–[18], [21], [22]. The results unanimously show that *perceived privacy* has a positive impact on the *attitude towards e-shopping* respectively the *consumer's trust in the online retailer* and online *purchase intentions* [5], [16], [17], [21]. *Perceived privacy concerns* in contrast have been shown to negatively influence online *purchase intentions* [18], [22].

Since anonymity – as it ensures privacy – is a measure to address privacy concerns, we argue that from a consumer perspective anonymous online shopping seems to be a desirable feature.

E.g. if a consumer is interested in "e-shopping", respectively has an online purchase intention, but is deterred from doing so by privacy concerns, the possibility to stay anonymous could resolve this dilemma. Accordingly, from the perspective of online shops, it might be interesting to give consumers the opportunity to shop anonymously from them, as this could attract new customers.

However, although the above shows that there is some support for this reasoning, we neither claim that the review presented in [3] is a systematic literature review nor that this is an empirically proven conclusion (for details, see [3, Sec. 3.5]). Instead, the review's findings basically support an assumption on which the presented research is based.

Table 12 in Appendix A gives an overview over the reviewed sources. For more details on this part of the review please refer to [3, Sec. 3.5].

2.2. Demand for All-Steps Anonymous Purchase Platform

In our earlier literature review we not only searched for support regarding if there is a need for anonymous online shopping, but we also searched the literature for architectures that "*would enable consumers to anonymously purchase goods online*" [3, p. 6]. Hereby, we focused on designs that cover steps[2] 1 to 4 of an online purchase as described in Section 1.1.1. The optional step 5 (returns) was considered, too, but not described as a separate step therein. For architectures which accomplish this we created the term *All-Steps Anonymous Purchase Platform* (ASAPP). Additionally, the review considered several papers which described technologies that seemed to have the potential for being combined or extended to designs of *All-Steps Anonymous Purchase Platforms.* The reason for focusing on architectures like this was our assumption that consumers would prefer to go through all steps of the purchase process using a single website/platform instead of having to use a variety of tools or services [3, p. 6].

In summary, the review included 7 designs which in combination cover all steps of anonymous online shopping [2], [4], [6], [7], [23]–[25], with 2 designs covering all steps by themselves [2], [6]. Four designs used known and partially extended cryptographic primitives [4], [6], [7],

[2] Called "phases" in [3].

[24] and three designs were based on fundamentally different approaches [2], [23], [25]. Since most designs had a high [4], [7], [23], [24] or even very high [6] level of detail we came to the conclusion that in general it seemed to be possible to implement some designs in the real world (outside test environments), e.g. combining several designs to cover all steps where necessary; however, to the best of our knowledge no such implementation was performed [3, p. 9].

A possible reason for that is that a real-world implementation requires market research on which a viable business model can be built. It is possible that this simply has not been done, yet, or that the market research showed that there is no viable business model for the respectively considered design. However by only considering the designs themselves we assume that a more fundamental reason for the lack of such an implementation lies in the drawbacks and limitations of the reviewed designs: Several designs did not consider or allow for the optional 5th step of returning items or performing criminal investigations, which, since it is a legal requirement, makes them insufficient for a real-world implementation [7], [24], [25]. One design actually did not provide complete anonymity and therefore could not be used – not even in combination with other designs – unless significant changes would be made to it [23]. Two further designs obviously required, among others, a high [4] or even very high [6] integration effort for the parties involved, making it unlikely that they could be implemented in reality within a reasonable time [3, p. 9].

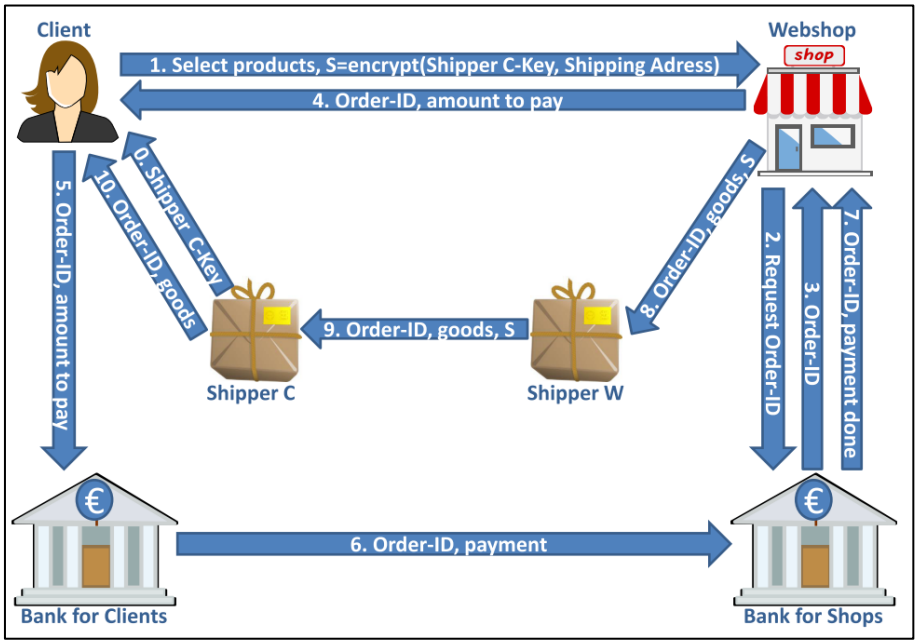

Figure 3: Overview of the approach to anonymous shopping in the Internet [2, Fig. 1]

The remaining design – [2] – had a somewhat simple but nevertheless very interesting approach as it was based on the concept of *Separation of Data* (see Figure 3). However, it was

also not realizable right away since it was not very detailed, required a high integration effort for multiple parties and followed a questionable approach regarding the handling of criminal investigations via merging of data. Additionally, as can be seen in Figure 3, the design used a common order id throughout all parties involved. Although in this design these parties are legally and physically separate, based on this order id in general it would be very easy to link a certain order, respectively certain products, to a certain customer in the case that two or more parties would maliciously cooperate.

Table 2: Concept matrix of technical designs with highlighted drawbacks/limitations (modified version of [3, Fig. 1])

Reference	Type of work	Approach	Level of detail	Assumed integration effort for involved parties	Anonymized steps/concepts						
					Browsing/selecting goods	Specification of payment/ shipping information	Payment authorization	Delivery/receiving of goods	Return of goods	Criminal investigations	Other
Androulaki, 2011 [6]	THES	Identity Management Model	very high	very high	+	+	+	+	+	o	+
Bella, Giustolisi, & Riccobene, 2011 [23]	JOUR	usage of cover data, balancing of anonymity and trust	high	middle	o	-	-	-	-	-	
Groppe, Kuhr, & Coskun, 2018 [2]	JOUR	Separation of Data	low	high	+	+	+	+	+	o	
Li, Zhang, & Zhang, 2016 [7]	CONF	Attribute based encryption & hidden access tree	high	middle	-	o	o	+	-	-	+
Rennhard, Rafaeli, Mathy, Plattner, & Hutchison, 2004 [4]	JOUR	Pseudonymity components	high	high	+	o	+	-	-	+	+
Sasson et al., 2014 [24]	CONF	Improvement of Bitcoin	high	high	-	o	+	-	-	-	
United States Patent No. US 2001/0029472 A1, 2001 [25]	PAT	Terminal based design, implicit separation of data	middle	middle	+	+	o	o	-	-	

Yet, our impression was that the design's general approach of providing anonymity by the *Separation of Data* nevertheless could be used to design a realizable *All-Steps Anonymous Purchase Platform:* If it was extended (and detailed) in a way that allowed for *Separation of Data* within a single platform, this would overcome its limitations regarding the detail, the

integration effort and the handling of criminal investigations [3, p. 9]. Additionally, the modified design must not require using a common order id – which would reduce the risk of successful correlation attacks by two or more maliciously cooperating parties.

As previously stated, we do not claim that our literature review presented in [3] is a systematic literature review. Thus, we have essentially only found support – but not proof – for our assumption that there is a demand for anonymous online shopping. Also, we identified designs that aim at providing this service but apparently have not – and/or cannot – be realized in the near future without modifications for various reasons. Finally, we argue that the review supports our reasoning to select the design by Groppe et al. [2] and then to propose an extensively modified and extended design – overcoming the original design's shortcomings – that can be efficiently implemented in the real world (for details, see [3, Secs. 3.6-3.7]).

Table 2 gives an overview over the reviewed technical designs. For more details on this part of the review please refer to Table 13 in Appendix B and to [3, Secs. 3.6-3.7].

2.3. Recent related work

In the previous sections (2.1 and 2.2) we showed that our earlier literature review [3] motivated this thesis by finding support for our assumption that there is a demand for anonymous online shopping but that there is a lack of designs that are deployable in the real world. In this section we want to briefly discuss recent related work:

We searched for related work first by using search engines such as the University of Ottawa library's *Omni* and *Google Scholar*. Then, we searched scientific databases like *Scopus* (scopus.com) and *Web of Science* (webofknowledge.com) by searching literature containing terms such as "anonymous" in combination with "online shopping" and multiple pseudonymous terms for both.[3] In addition we also searched for recent citations of the

[3] E.g. search in *Web of Science*: TI=((anonym* OR privacy-preserving OR pseudonym* OR disguised) AND (e-commerce OR ecommerce OR online shopping OR web shopping OR shopping in the internet OR internet shopping))
e.g. search in *Scopus*: TITLE-ABS-KEY((anonym* OR privacy-preserving OR privacy OR pseudonym* OR disguised) AND (e-commerce OR ecommerce OR online shopping OR web shopping OR shopping in the internet OR internet shopping))

literature that we had reviewed in [3] and also revisited their references. In the case of numerous results, we filtered to current years (since 2018) but we also included a few earlier works that we had not found before.

Overall, we have found ten related publications (seven recent, three (slightly) earlier), with two being systematic literature reviews regarding “personal information privacy concerns”[26] respectively “Information disclosure in e-commerce” [27], one doctoral thesis that discusses “Privacy-preserving e-commerce protocols” [28], three studies on different aspects of anonymity and privacy in the context of online shopping [29]–[31] and four papers that propose designs of systems that realize different kinds of anonymous e-commerce platforms [32]–[35].

Regarding the systematic literature reviews presented in [26] and [27] we note that they largely close the research gap that we identified in our earlier literature review, calling for “comprehensive literature review[s] of information privacy research” [3]. Their focus is, however, narrower because they consider “personal information privacy concerns”[26] and even more specifically “Information disclosure in e-commerce” [27]. We also want to mention an additional earlier review – [36] – which, however, only includes two publications that are related to anonymous e-commerce, one of them being discussed below.

In contrast to the earlier thesis by Androulaki [6] the more recent thesis by Rial [28] is not a comprehensive proposal of a new “Privacy Preserving ECommerce Oriented Identity Management Architecture” [6] but mainly a summary of Rial's earlier work on "Privacy Preserving E-Commerce Protocols" [28]. This work included applications that are less related to our thesis, such as “Smart Metering” and “Toll Pricing” [28] but Rial also discusses *Private Purchase Protocols* the property of which is that the seller obtains knowledge about the identity of the customer but not about the products the customer purchases and *Anonymous Purchase Protocols* whose property is the exact opposite.[28] Interestingly, Rial suggests to combine these protocols to create a protocol that provides both privacy and anonymity. In a nutshell, this is what our proposed platform does, but by different means – namely the *Separation of Data* – so that the *Frontend Company* provides privacy, and the *Backend Company* provides anonymity. In combination, our proposed platform protects the customer’s

privacy and provides anonymity, however, admittedly, not in the strict sense that Rial's suggestion would presumably achieve.

As we noted in Section 2.1, in our earlier literature review [3], we did not find any research that explicitly examines if there is a need for anonymous online shopping but we were able to find support for our assumption that there is a demand for anonymous online shopping, nonetheless. Fortunately, the three studies [29]–[31] are even closer to explicitly examining a potential need for anonymous online shopping and they support our earlier findings discussed in Section 2.1: Amongst others, Chen et al. [31] show, by means of a survey, that "a certain degree of anonymity […] encourag[es] constant purchases" [31, p. 14]. Regner and Riener [30] examined real-world data to show that the revenue of a Pay-What-You-Want (PWYW) music shop dropped by about 25% when the identity of the customers was revealed to the artist instead of being kept secret. The reason for this was a reduced number of customers purchasing due to – as they suspected – privacy concerns. They verified this finding by means of an online experiment, where revenue dropped by 35%. [30] Finally, the study of Bandara, Fernando and Akter [29] showed that the reason for consumers providing their personal information despite having privacy concerns (the *Privacy Paradox*) can be related to a state of "learned helplessness" [29]. In essence, this means that these consumers provide personal information since they believe they have no choice, but they do so reluctantly and might feel "passive, withdrawn, and dissatisfied" [29]. We argue that in contrast to this, a platform that provides the possibility to shop online anonymously, could be used by those consumers with more positive feelings.

Finally, the four papers that propose designs of anonymous e-commerce architecture can be categorized into two groups: [32] and [33] utilize blockchain technology for their peer-to-peer marketplace designs that allow for delivery of electronic and physical goods, whereas [34] and [35] are designs for online retailing which are based on other technologies and only allow for delivery of electronic goods. Regarding the two blockchain-based P2P marketplace designs it should be noted that [32] is more of an idea sketch which also does not seem to be practical, as it fails to consider, for example, how physical goods can be delivered while maintaining anonymity, and also requires the use of a marketplace-specific currency called "trade coins". In contrast to this [33] provides extensive detail, including how payments can be made in USD

and how the anonymous delivery of physical goods can be realized by utilization of *Internet-of-Things (IoT)*-enabled logistic centers and delivery boxes. When examining the two online retailing designs more closely, we note that the design of a "Fair Exchange and Anonymous E-Commerce" by Mars and Adi [34] seems to be realizable in general but to require a high integration/deployment effort, since for example customers and retailers need to purchase and use clone-resistant *Commercial Hardware Tokens (CHT)* and must also involve a trusted *One-Time Coin Generator (OTC)* and *Electronic Commerce Server (ECS)* into the purchase and payment processes. In contrast to this the design of a "*Privacy Preserving E-Commerce Protocol (PPEP)"* by Ike and Sarac [35] seems to be more practicable as it only requires the installation of software at customers and retailers and the involvement of one additional party, called PPEP broker. We want to point out that this design – [35] – is the closest to the design proposed in this thesis, as it, for example, similarly establishes an encrypted communication channel between retailers and customers via a third party that acts as a proxy in order to disguise the customers' meta information such as IP addresses and browser type. However, as we will show in the following, our proposed design has the advantage of allowing for the delivery of physical goods and not requiring the installation of additional software neither at customers nor retailers (transparent integration). Also, in the design presented in [35], the broker obtains information about which customer purchased products for a certain amount from which retailer. Furthermore, the retailer can – by default – attribute separate purchases to one anonymous customer with the authors apparently inadvertently misstating the property of *Transaction Unlinkability*.[4] As we show in sections 6.2.1 and 6.2.2 these two drawbacks are not present in our design.

[4] However, the authors show that real *Transaction Unlinkability* can be achieved in [29], too.

3. Objectives

3.1. Platform Architecture Design

As described in Section 1.2 the first objective of this research is to design a new architecture for an *All-Steps Anonymous Purchase Platform*, based on the concept of *Separation of Data* as proposed in [2], but overcoming its limitations by realizing this concept within a single platform. An alternative description of such a design is to see it as a kind of escrow service which is enhanced by the concept of *Separation of Data*: The platform acts like an escrow service to the parties involved (customer, online shop, bank, shipper) but the internal separation of data between a split-up *escrow service frontend company* and an *escrow service backend company* ensures that neither online shop, bank and shipper nor individuals at these escrow service companies can link the customers to the products they purchased. Figure 4 shows the simplified data flows for an online shopping process using escrow services.

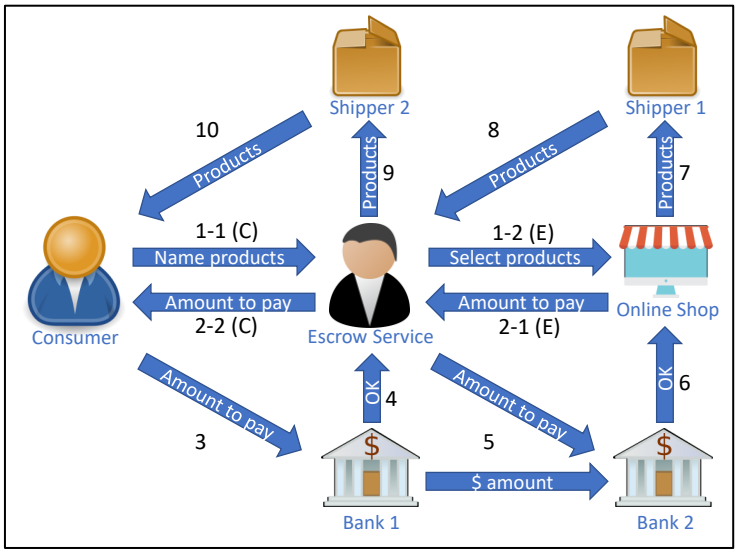

Figure 4: Simplified online shopping process data flow using escrow services

Figure 5 is an early architectural blueprint of the proposed design. It

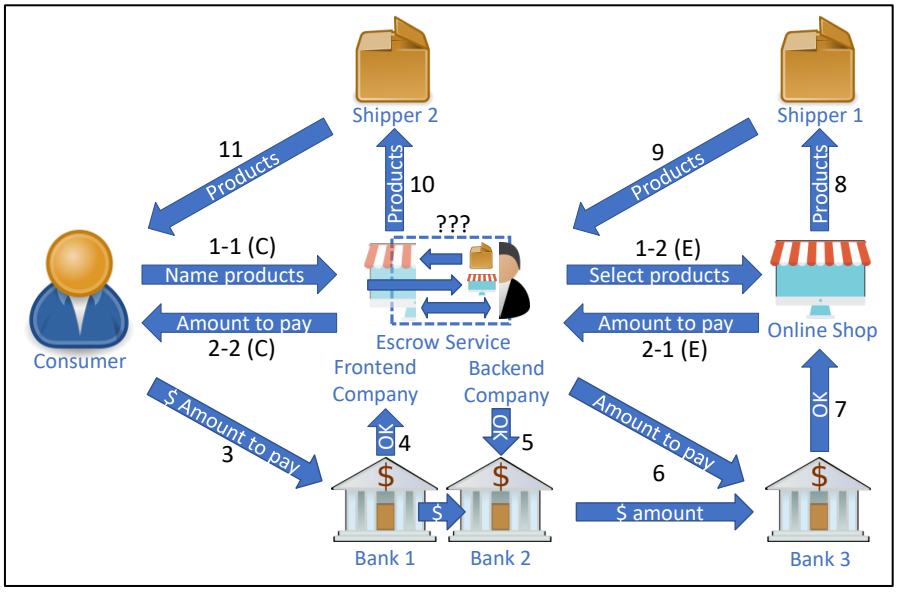

Figure 5: Simplified online shopping process data flow using escrow services with internal separation of data

adds the concept of *Separation of Data* to the internal workings of the escrow service. To work out the details of the internal workings and the overall architecture is the first main goal of the research presented here, including outlining how physical products are handled between the escrow service frontend company and escrow service backend company. The remainder of this section will describe the objectives of this part of the thesis in more detail while at the same time pointing out the improvements over other approaches and previous designs.

From the consumer's point of view, the platform shall provide anonymity and be convenient to use since it covers the complete purchase process including browsing and selection of products, payment, delivery and returns. The combination of these features is an improvement over the tools and services mentioned in the introduction ‑ including standard escrow services ‑ and the designs [4], [7], [23]‑[25] discussed in Section 2.2.

For the involved web shops, banks and shippers the platform may not require additional integration effort which is an improvement over the aforementioned designs. Of course, it does require effort to implement the platform and it does require integration effort to connect it to banks, shippers and online shops but this may not be higher than for other customers of these institutions.[5] From their point of view, the platform should behave like a normal (business) customer, for example using already existing GUIs, web services or other APIs.

Regarding the internal workings of the platform, it must implement an automated purchase handling so that the administrative overhead for each individual purchase stays within a reasonable range. This is an improvement e.g. over escrow services that handle customer requests on an individual basis. At the same time the platform has to allow for proper handling of criminal investigations without the necessity of manually merging data provided by potentially numerous parties. This is another improvement to the approach proposed by Groppe et al. [2].

Overall, the platform's architecture must be designed in a way which realistically allows for implementing it in the real world: it must be convenient to use (single point of access for

[5] Optionally, agreements could be made with online shops on discounts or commissions that could be used to support the financing of platform operation. Similarly, agreements could be made with banks and shippers on discounts to reduce platform operation costs. See Section 6.4.4.

consumer), provide anonymity respectively privacy for all purchase steps (see Section 6.2.1), adhere to practical and legal requirements and it must not require a high integration effort for the parties involved (consumer, shippers, online shops, banks). Naturally, it also has to work efficiently in order to be implementable in the real word. Chapter 6 will discuss criteria and means with which we validate whether these objectives have been met.

3.2. Platform Prototype Implementation: GhostBuy

In addition to detailing the architecture of the platform and describing important workflows the second main goal of this thesis is to demonstrate/evaluate the feasibility of the to-be-found architectural design by implementing a working prototype. Before March 2020 we planned a very open-ended approach to the design evaluation, in fact even considering to simulate the data flows of the design only manually. However, several reasons, including early insights into how to solve certain problems, already having ruled out several technologies for various reasons (such as inefficiency), the *COVID-19 pandemic* and discussions regarding possible thesis contributions, narrowed our goals. In particular, we found that a major contribution of the thesis would be the implementation of a prototype as opposed to only simulating the data flows. Therefore, we decided to implement a prototype for evaluating our architecture design as follows:

1. It was specified that the prototype would be implemented by adapting and extending the *Model View Controller (MVC)* pattern-based architecture of a bookshop web application demo on which we worked for a previous student project. The prototype should cover the following:

    a. Browsing, selecting, ordering & paying for goods from a real-world online store / e-commerce website (might use "insecure" prototype payment option such as e-Transfer)

    b. Payment forwarding from frontend to backend (interface and prototype implementation)

    c. Separation & anonymous exchange (matching) of data between frontend and backend

   d. Handling of unwanted information leaking (in communication between backend and customer)

2. Regarding the implementation we committed to evaluate if a purely browser-based approach would be a reasonable way of implementing the client-side application for the customer – at least as a prototype.

3. In addition, we committed to provide workflow descriptions (e.g. for handling physical products, e.g. handing them over from backend to frontend) to outline how these could be realized.

4. Depending on how the COVID-19 related situation would develop after March 2020 and also depending on how soon we would be able to implement the above, we optionally planned to add:

   a. Hash-based verification of transactions

   b. QR code or alternative verification of received packages (package seal)

   c. Query interface(s) for law enforcement authorities

5. In case that there would be not enough time to implement the latter three (4a, 4b, 4c), we pledged to outline how this could be achieved in order to show that we identified the related research problems and that there would be solutions to them.

The decision to adapt and extend the bookshop project's architecture was made in order to save time during the design and implementation of the prototype since in general we already were familiar with the technologies used therein. However, even at this early stage of the research it was clear to us that for this thesis we nevertheless would have to write almost all code from scratch. In addition, technologies that were not part of the bookshop project had to be incorporated in the prototype's design and implementation.

3.3. Evaluation of Objectives

The evaluation of the thesis' results respectively the evaluation of the architectural design and the implemented prototype will be performed using the criteria *Privacy & Anonymity*, *Security* and *Deployability*. The criteria are described in detail in the respective paragraphs of Chapter 6.

4. Research Problems, Methods, Process & Important Decisions

The following describes our iteration of the *DSR* process steps *Design & Development* and *Demonstration* (see Figure 2) and certain important decisions we made during that process:

We started with the first architectural blueprint described in Section 3.1 (see Figure 5) and defined the data flows and the required cryptographic properties – as far as they were already identified – in more detail. In particular, we identified five main research problems that would need to be solved by the architecture. For this we named the escrow service frontend company the *Frontend Company* and the escrow service backend company the *Backend Company*. Additionally, there is a third party involved, the *Customer*.

1. *The Frontend Company* and the *Backend Company* each have to be able to refer to the *Customer* respectively to data linked with the *Customer* (e.g. customer's order) without any employee of the *Frontend Company* being able to collaborate "offline", i.e. separate from platform`s secure environment, with any employee of the *Backend Company* – and vice versa – to match/link datasets about the *Customer* stored in each company's database.

2. If employees of the *Frontend Company* or the *Backend Company* try to (maliciously) collaborate to match datasets about the *Customer* this behaviour should be detectable.

3. When the *Customer* has to transmit information to the *Backend Company* or vice versa, this information must not be obtained by the *Frontend Company*. At the same time, the type of information being exchanged between the *Customer* and the *Backend Company* must be controllable in order to avoid certain types of personal identifiable information being exchanged between them. E.g. the *Backend Company* may not obtain the name or IP address of *C*.

4. The platform (*Frontend Company* and *Backend Company*) must have a means of communicating with the *Customer* regarding any previous transaction even after the direct interaction – in which this transaction took place – with the platform ended (e.g. the *Customer* closed the browser and the *web session* ended). Unlike practiced by certain existing online retailers, sensitive information may not be sent via e-mail even

when the e-mail is encrypted in transit (e.g. since it may me readable by the email provider and would be popping up in plaintext in the customer's email program).

5. The platform must be able to provide data for criminal investigations. However, the interface to do so should provide the minimum required information. I.e. that providing such data should be done in a way which does not allow (or force) neither the *Frontend Company* nor the *Backend Company* to learn who/what was investigated. This is to keep the anonymity of customers while still being in line with the law.

Overall, we identified 19 research problems but the aforementioned five were the most crucial to solve. Table 17 in Appendix U provides a complete list of identified research problems alongside with information on how we addressed them.

Second, we searched the literature for cryptographic schemes and systems that could potentially be used to build this architecture. We had planned to perform a broad search using scientific databases like *Scopus* (scopus.com) and *Web of Science* (webofknowledge.com) but also using rather general-purpose search engines such as google, the University of Ottawa library's Search+ and google scholar. The initial keywords we planned to use were based on the necessary cryptographic properties we defined in the beginning and also potentially similar application areas of cryptographic systems. They included but were not limited to keywords such as *(threshold) key escrow*, *zero knowledge proof of knowledge*, *time lock puzzles*, *homomorphic encryption*, *blockchain* and *smart contracts*. We had planned to subsequently design several potential architectures that would implement the architectural blueprint and the necessary data flows as defined in the beginning, assess them, review the first blueprint, redesign/refine the architecture and so on until reaching a self-chosen time limit for this research phase. At that point – we had planned – we would decide on how to evaluate our architecture, e.g. based on a simulation or prototype implementation and afterwards perform this evaluation.

However, as can be seen from the previous Sections 3.2 & 3.3, already at an early stage of the thesis we focused on certain approaches and had ruled out other approaches regarding how to solve our main research problems. Therefore, our literature research focused on cryptographic algorithms and protocols such as *Oblivious Transfer (and Extensions)* [37]–[40], *Commitment*

Schemes and Zero-Knowledge Protocols [41], [42] and the related concepts *Matching Oblivious Transfer* [43], *Oblivious Hashing* [44] and *Private Set Intersection Based on OT Extension* [45]. We found that especially *Oblivious Transfer* and *Private Set Intersection* are technologies that could prove to be useful for implementing a data interface for law enforcement agencies. *Oblivious Transfer* was also considered as an alternative method for agreeing on one-way identifiers which is an approach we had considered for matching/linking datasets of the *Frontend Company* and the *Backend Company*. However, we ruled out *Oblivious Transfer* for this, since it did not seem to have an advantage over one-way functions such as hash-functions. In fact, in our prototype implementation we did not even use one-way functions for this specific purpose but instead found a better solution (see Section 5.6.9).

Please note that – as we now move from discussing general architecture requirements to the specific platform we are proposing – in the following the proposed platform, the prototype and its architecture will be called *GhostBuy Platform*, *GhostBuy Prototype* and *GhostBuy Architecture* or depending on context, simply *GhostBuy*. We would like to emphasize that the specific implementation of the *GhostBuy Prototype* (i.e. also our specific *GhostBuy Architecture* as detailed in Section 5.1) is, however, only one of probably many possible implementations that realize our proposed design shown in the architectural blueprint in Figure 5.

In addition to the above literature research we also searched for browser-based implementations of cryptographic functions that could be used for implementing the client-side functionality of our proposed platform prototype, the *GhostBuy Prototype*. At first glance it might seem to be the wrong approach to search for implementations of cryptographic functions before having known which exact functions we would want or need to use within the *GhostBuy Prototype*. We decided to do this nevertheless, since we were sure that we would not have enough time to implement cryptographic functions that we wanted to use in the prototype from scratch ourselves. Our search yielded only two promising implementations from reputable sources: The *Stanford Javascript Crypto Library* [46]–[48] and the W3C's *Web Crypto API* [49]. We decided to use the *Web Crypto API* since – despite being a JavaScript API – it efficiently runs the cryptographic functionality natively in the browser and has been reviewed by a large number of people, including scientific reviews such as [50].

Additionally, it has been implemented in the vast majority of current browsers – including mobile browsers – which is a requirement since it is not a JavaScript library that can be simply used in any browser supporting JavaScript [51], [52]. Finally – in contrast to the *Stanford Javascript Crypto Library* which recently seems to have lost public attention and community support – the *Web Crypto API* is still being developed and maintained on a regular basis [46], [53].

Once we had settled on the *Web Crypto API* the next step in our research was to develop the *Initial Cryptographic Setup* that the *GhostBuy Prototype* would have to create for all customers browsing its website, restricting ourselves accordingly to the most secure, suitable algorithms supported by the *Web Crypto API*. Despite not yet being final, the setup already envisaged a layered architecture of session-specific encryptions, through which the *Backend Company* and the *Customer (C)* would exchange encryption keys, and also a concept that we dubbed *Virtual Session*. The setup moreover already included storing session-related encrypted information in the browser-local *IndexedDB* and making use of a client-side key for later retrieval of previous purchases. However, during the implementation, slight changes were made to the setup. The details of the setup are described in Section 5.5.

At this point in our research, we started implementing the actual *GhostBuy Prototype*, beginning with the realization of the *Initial Cryptographic Setup* and from thereon one-by-one adding (and testing) the required functionality for each step a customer would sequentially perform when using the prototype. This iterated process (design, implement, demonstrate, evaluate) also yielded minor architectural changes (see Section 5.1) and implementation revisions (see e.g. Section 5.6.6).

While iterating through these steps we documented the research's process and progress for later inclusion in this thesis, especially by keeping a development logbook and commenting and documenting the source code of the prototype implementation. This contributed to accounting for the last step, *Communication*, of the *DSR* process (see Figure 2: →*Communication*).

5. Prototype Design & Implementation

5.1. Preliminary and Final GhostBuy Prototype Architecture

The preliminary architecture of the *GhostBuy Prototype* has been adapted from the web application architecture of a simple bookshop on which we worked as part of a previous student project. The architecture adheres to the *Model View Controller (MVC)* pattern in order to separate the business logic and business-internal representation of information from the way it is being presented to the customer and from how the customer can interact with that presentation. A simplified architecture of an MVC web application can be seen in Figure 6.

To realize the intended platform-internal *Separation of Data,* the bookshop project's

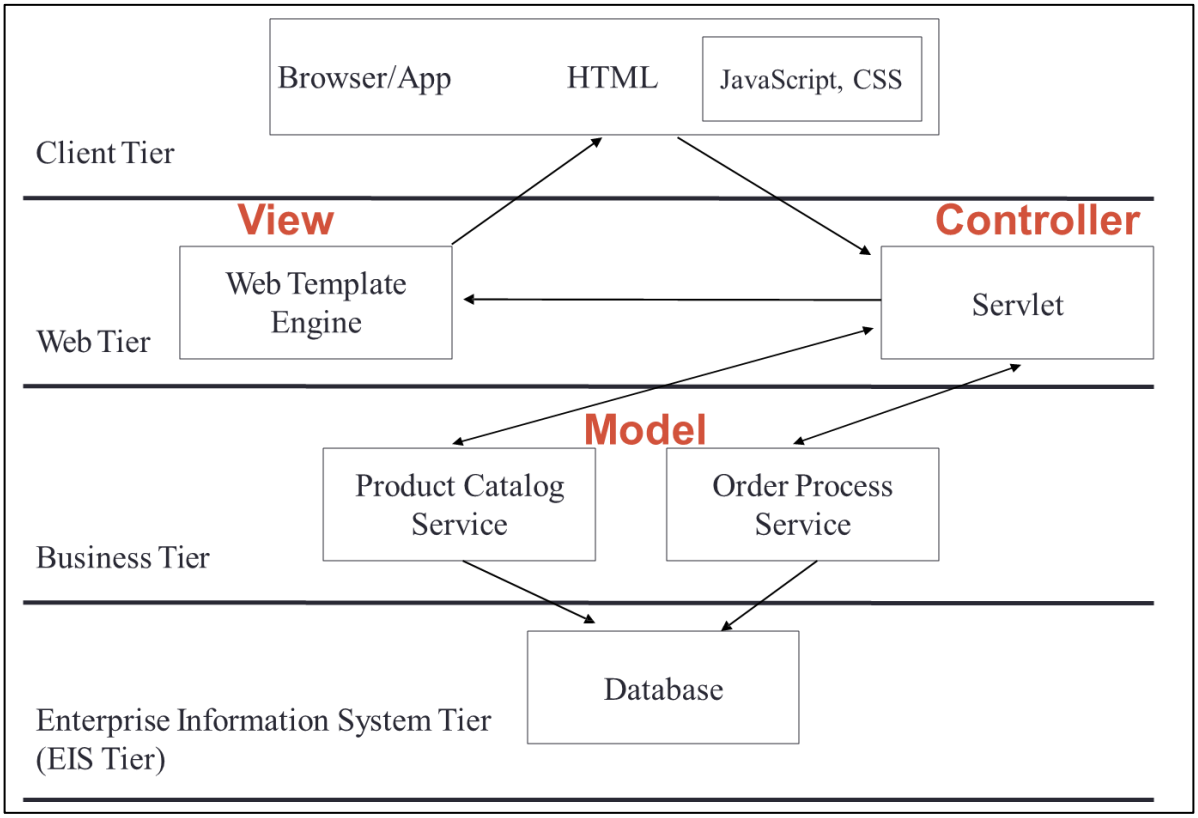

Figure 6: Simple MVC web application (modified version of [70, p. 17]*)*

architecture was extended by splitting the *Model (*consisting of the *Business Tier* and *Enterprise Information System (EIS) Tier)* into a *Frontend Domain* and a *Backend Domain.* I.e. – despite the name – both domains are part of the business logic that is often referred to as "the Backend" in web applications. Thus, in the preliminary prototype architecture – and also in the final architecture – the *Model* is implemented by two applications that we call *Frontend*

Application and *Backend Application.* However, in other web applications the application that implements the *View* and the *Controller* – i.e. the application the customer is directly interacting with – might be similarly called "the Frontend". Therefore, to avoid confusion, this application subsequently will be called *Portal Application* throughout this thesis. Figure 7 illustrates the *GhostBuy Prototype's* preliminary architecture with its three applications as also described in the following.

As can be seen from the highlighted blue areas in Figure 7 the preliminary architecture consisted of three independent, but linked applications, being the *Backend Application*, the *Frontend Application,* and the *Portal Application.* In addition, the customer's web client (e.g. browser) executes *JavaScript* code that is provided by the *Portal Application* and thus can be seen to be the client-side part of that application. However, to avoid confusion we call it *Client Application* – and rarely the *Portal Application's client-side functionality* – throughout this thesis. The *server-side Portal Application* is simply called *Portal Application* unless we need to differentiate it from the *Portal Application's client-side functionality*.

In a real-world implementation the *Backend Application* would be run by the *Backend Company*, and the *Frontend Company* would run/provide the other two applications (respectively three applications, if the *Client Application* is counted as a separate application). We depicted the separate company infrastructures with grey areas in Figure 7.

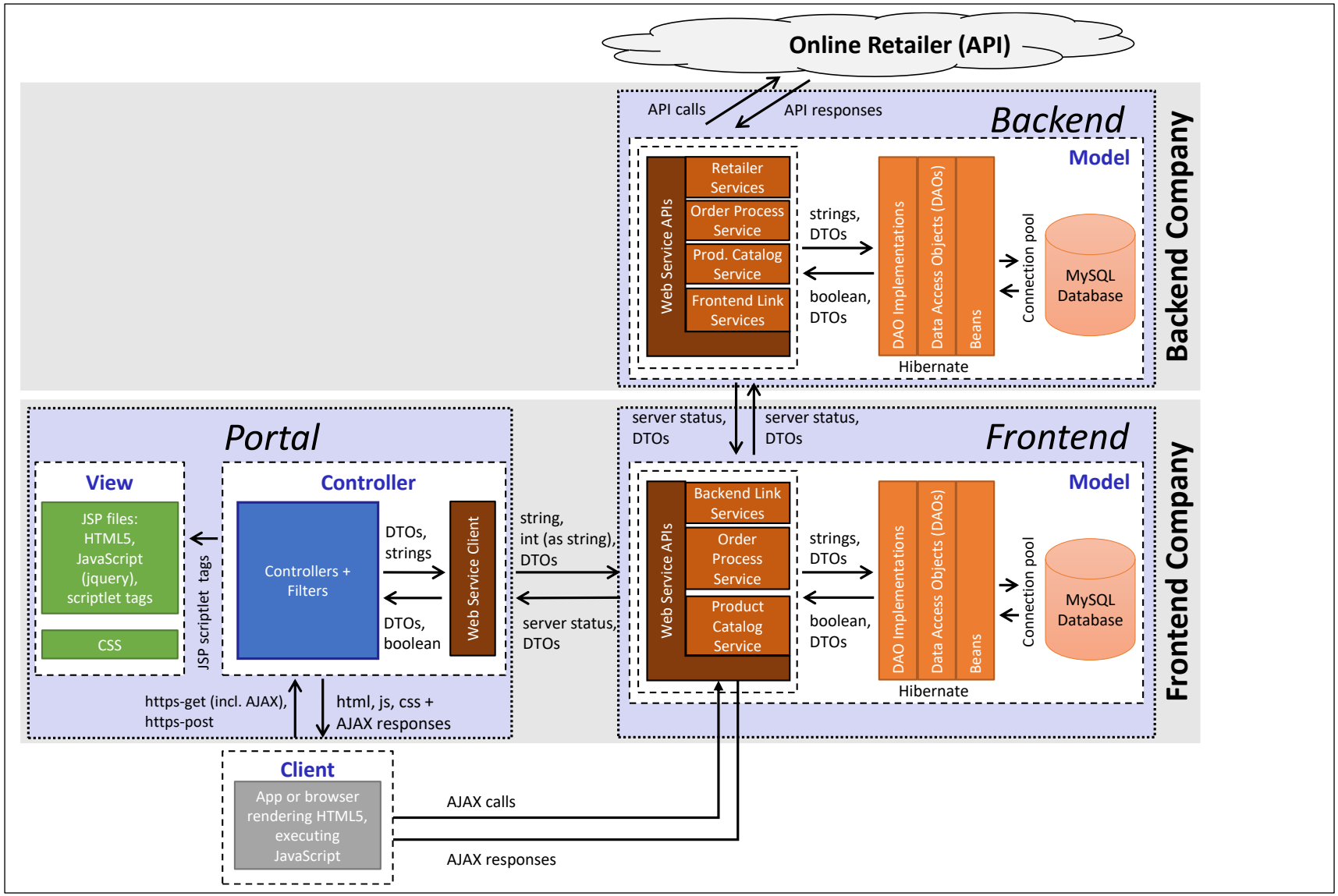

Figure 7: Preliminary architecture of GhostBuy Platform Prototype implementation

Although the *GhostBuy Prototype's* preliminary architecture was partially modified during the implementation it already incorporated the following important concepts and technologies:

- The *Frontend Application* does not exchange information with online retailers / e-commerce websites, only the *Backend Application* does. The reason for this is that the *Frontend Application* must not obtain information about which product information a customer searched for, respectively which products the customer finally purchased – or not.
- The *Backend Application* does not directly exchange information with the *Client Application*, i.e. the customer, but only through the *Frontend Application* (in the final prototype mainly through the *Portal Application*). This is to control the type and the structure of data/information being exchanged between the *Client Application* and the *Backend Application* and thus also to prevent certain information about the customer – e.g. the IP address – from becoming known to the *Backend Application*.
- The *Frontend Application* and the *Backend Application* utilize separate databases to ensure that order-related information can be stored permanently but remains unknown to the other application/company. Naturally, there has to be a mechanism – e.g. some sort of identifier – for linking datasets from the *Frontend Application* and the *Backend Application* to a customer's order. Before the implementation phase, we considered using one-way functions to create this link. These functions would need to incorporate secret components in order to protect the link from getting known to adversaries. However, during the implementation phase we instead decided to place identifiers – encrypted by the respective other application – in the database of both applications. This concept is explained in more detail in section 5.6.9.
- The applications use a common set of Data Transfer Objects (DTOs).
- The *Client Application* partially uses *Asynchronous JavaScript and XML (AJAX)* calls to request information asynchronously/dynamically from the prototype's other applications. However, despite the depiction in the preliminary prototype architecture (Figure 7), for the actual implementation we refrained from making any direct calls (AJAX or otherwise) to the *Frontend Application's* web services but let all calls be

handled, i.e. processed and – where necessary – forwarded, by the controllers of the *Portal Application.*[6]

- The *Client Application* is realized in the form of a web client, i.e. browser, that renders HTML5 pages which are dynamically provided by the *Portal Application's* controllers. The pages are based on *Jakarta Server Pages (JSP)*, include *Cascaded Style Sheets (CSS)* to define layout classes and utilize JavaScript to implement the client-side functionality. JSP scriptlets, however, were only used to include otherwise redundant JSP page parts, such as the header and footer, into the main JSP pages. We did not use them for inserting Java code into the JSP pages due to reasons which we explain in Section 5.6.1.

While the *GhostBuy Prototype's* final architecture (see Figure 8) is broadly similar to that of the preliminary prototype, we want to detail two important changes that are not explained elsewhere:

- In order to be able to control the type and the structure of the information that the *Client Application* exchanges with the *Backend Application*, we had originally planned that the *Portal Application*'s controllers would never communicate directly with the *Backend Application* but only through the web services of the *Frontend Application.* To achieve this, we intended to implement certain web services twice, once in the *Frontend Application* and once in the *Backend Application.* Hereby, each web service would communicate with its counterpart via two "paired" web services called *Backend Link Services* and *Frontend Link Services*.

[6] The depicted direct calls to the web services were a remnant of the original book shop web application architecture.

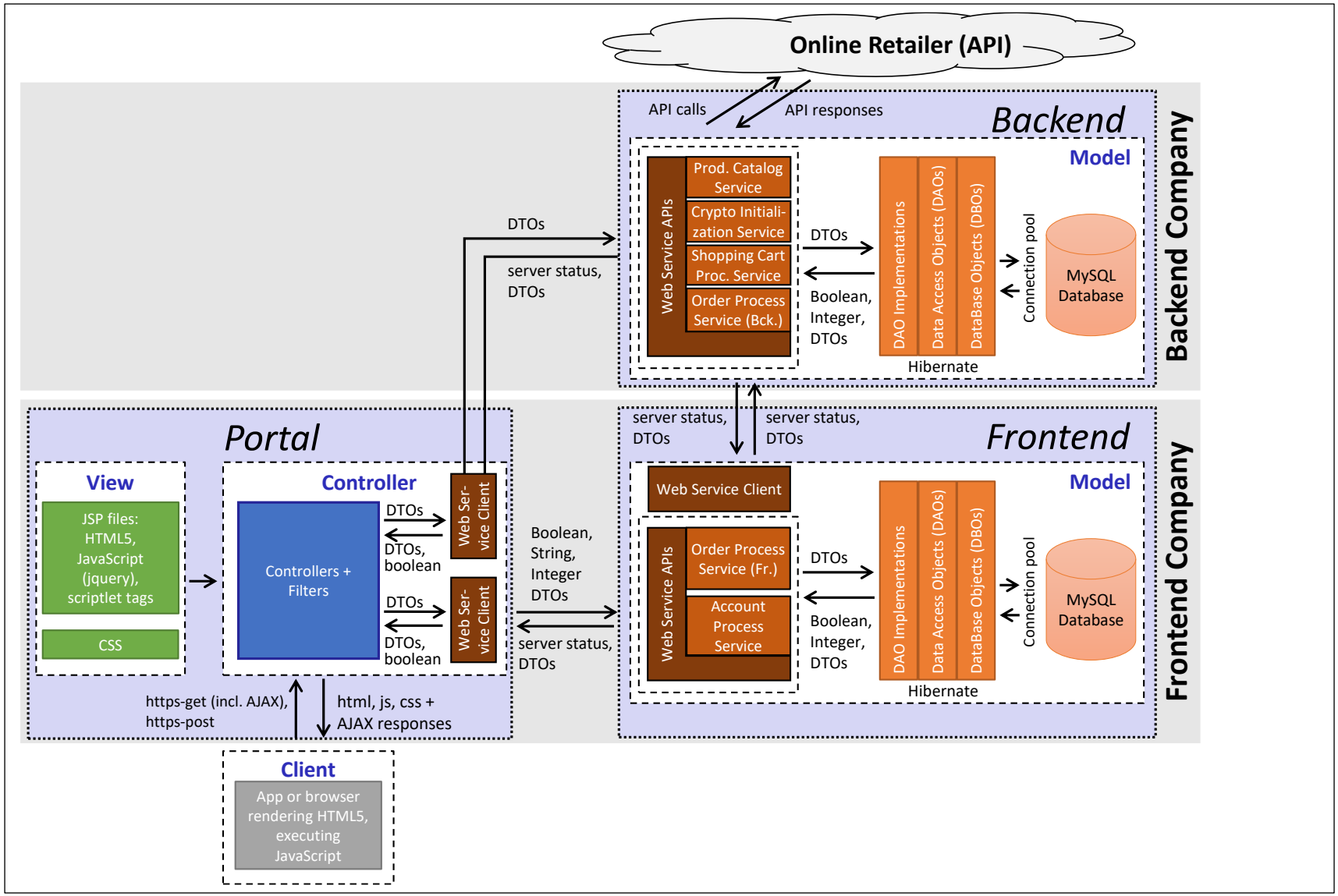

Figure 8: Final architecture of GhostBuy Platform Prototype implementation

However, during the implementation phase it became clear that there is no need to restrict the *Portal Application*'s controllers to only communicate with the *Frontend Application's* web services, since the type and the structure of information can also be controlled within the *Portal Application*. This meant that most web services only had to be implemented once, either in the *Frontend Application* or the *Backend Application.* E.g. the *Product Catalog Service* is implemented only in the *Backend Application* since it communicates with the e-commerce websites and provides the requested product information in encrypted form to the respective controllers in the *Portal Application*. The only web service that had to be implemented in both the *Frontend Application* and the *Backend Application* is the *Order Process Service.* This is because an order consists of information known only to the *Frontend Company* (e.g. the shipping address) and information known only to the Backend Company (e.g. the product details). Accordingly, the *Frontend Order Process Service* and *Backend Order Process Service* do not perform the exact same tasks but rather work together to perform the order processing (for details, see sections 5.6.8 & 5.6.9).

- The original bookshop architecture that was adapted for the prototype implementation incorporated direct AJAX calls from the client to the web services. Therefore, these calls were also depicted in the preliminary architecture diagrams. However, for the *GhostBuy Prototypes* final architecture and implementation we decided against any direct communication (AJAX calls or otherwise) between the *Client Application* and the *Frontend Application* – instead the communication happens only between the *Client Application* and the *Portal Application* – mainly for anonymity and security reasons: First, the concept of *Separation of Data* would have required that the direct communication took place between the *Client Application* and the *Backend Application,* as it would have served to exchange product information. However, as explained before, we must not allow any direct communication between the *Client Application* and the *Backend Application*, to prevent revealing information about, for example, the client's IP address or browser type. Secondly, we need to think of the *Portal Application* as the only internet-facing access point to the *GhostBuy Platform*, respectively that the *Frontend Application* and the *Backend Application* would be located in a more strictly protected network of the platform (behind additional

firewalls). Allowing for direct communication with the *Client Application* or any other internet-based device would therefore pose a security risk.

5.2. Overview of GhostBuy Prototype Development Environment

In our development environment we implement the *GhostBuy Prototype* through 3 projects that represent the *Portal Application* (which includes/provides the *Client Application* respectively the *Portal Application's client-side functionality*), the *Frontend Application,* and the *Backend Application*. To be able to run these applications on the same development-computer, each application uses a dedicated Tomcat v 9.0 server instance with each instance utilizing different ports. Since the *Portal Application* is the application with which the customer directly interacts, it uses the default https port 443 (http 80 is automatically redirected to https) and can be browsed locally using the URL *https://www.ghostbuy.xyz/*.

As illustrated in Figure 8, the *Frontend Application* and *Backend Application* implement the web services and database functionality that represent the split *Model* of the *GhostBuy Prototype's* extended MVC pattern-based architecture. An application-specific web service client is being used to access the respective application's web services.

The *Portal Application* implements the controllers, and servlet filters that dynamically provide the HTML pages and other web resources (*CSS files, JavaScript modules*) to the customer's web browser. Most of the *Portal Application's client-side functionality (Client Application)* – including the prototype-specific cryptographic functionality that is based on the *Web Crypto API* – is used on multiple pages. To avoid having redundant code, we implemented a JavaScript module that encapsulates this functionality in accordingly structured namespaces. This module, named *ghostbuy_module.js,* exports the necessary constants and functions so that they can be used and triggered by the other – webpage-specific – JavaScript modules.

The web pages that are provided by the *Portal Application* are based on a web page template called *OneTech* that we obtained from *colorlib.com.* The template includes some supporting *JavaScript* functionality, e.g. to initialize a drop-down menu. It must be understood that we strongly modified the template's layout and implemented all non-static website functionality. I.e. the template itself only contains static placeholder content, including static product descriptions, prices, images, and even a static number of "items in cart". The only exception

to this is the in-page filtering and sorting functionality that we nevertheless needed to modify for our purposes as described in Section 5.9 (to support in-page sorting and sorting of all products).

The communication between the three web applications takes place using TLSv1.2 connections. The same applies to the communication between the customer's browser, i.e. the *Client Application*, and the *Portal Application*.

The *Frontend Application* and *Backend Application* permanently store account information and order information by saving encrypted Java objects (that we dubbed *DataBase Objects (DBO)*)[7] in dedicated MySQL instances. For this, *Hibernate*[8] has been used to implement the *Object-Relational Mapping (ORM)*.

In addition to the three web application projects, the development environment contains two further projects in which we implemented the *DTOs* and the objects that provide the cryptographic functionality. These two projects, respectively the objects they implement, are used across all three web applications. This ensured that all web application projects always used the very same (and latest) version of the DTOs and cryptographic objects.

The top-level project structure as configured in the development environment (Eclipse IDE for Enterprise Java Developers 2020-03) that we used to implement the *GhostBuy Prototype* can be seen in Figure 9.

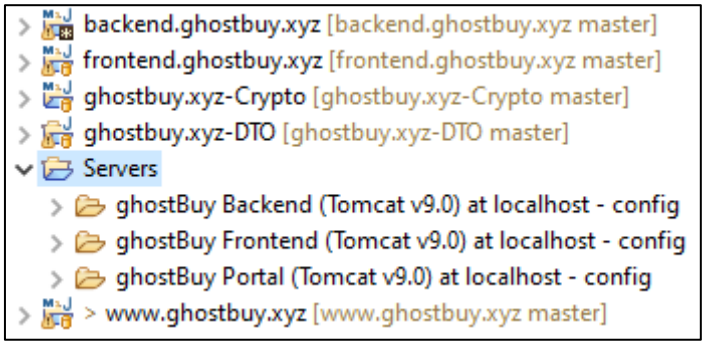

Figure 9: Top-level project structure for prototype implementation

A more detailed description of the *Prototype Development Environment,* that e.g. details the URLs, ports and names of the web applications and server instances, is given in Appendix C. We provide the source code, source code documentation and implementation description in Appendix W.

[7] In the context of Hibernate/ORM elsewhere often referred to as "beans".

[8] See https://hibernate.org/

5.3. Pages & Navigation of GhostBuy Prototype

Before we describe the actual prototype implementation, we want to give an overview of the *GhostBuy Prototype's* most important web pages and how they can be navigated, respectively what their respective purpose is. This is to help the reader get a better understanding of the web pages that are mentioned in the implementation description. We specify each pages' name, the URL (of its respective controller), the name of the actual JSP file (to which the controller forwards) and shortly describe its purpose. The order of the described pages follows a possible browsing flow of a new customer as illustrated in Figure 10.

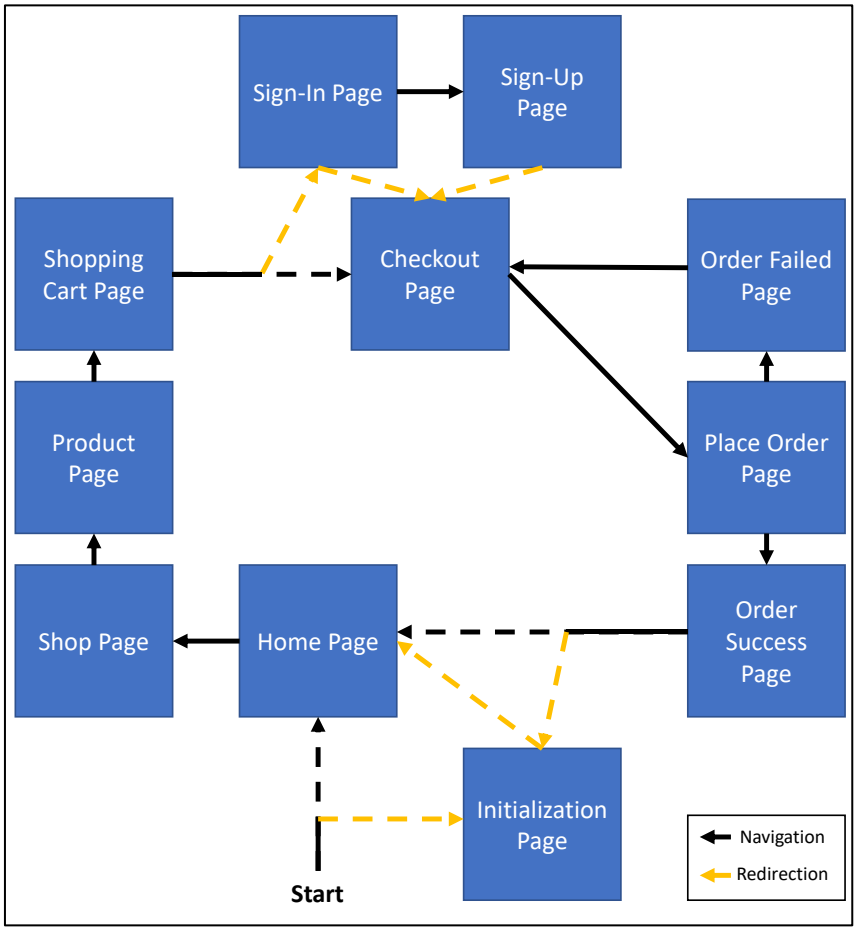

Figure 10: Example browsing flow of GhostBuy Prototype customer

In addition, we created two demo videos (overview demo & detailed demo) that show the *GhostBuy Prototype* in action. These videos can be found in Appendix W.

- ***Initialization Page*** *(https://www.ghostbuy.xyz/Initialize➔InitializeCrypto.jsp)*

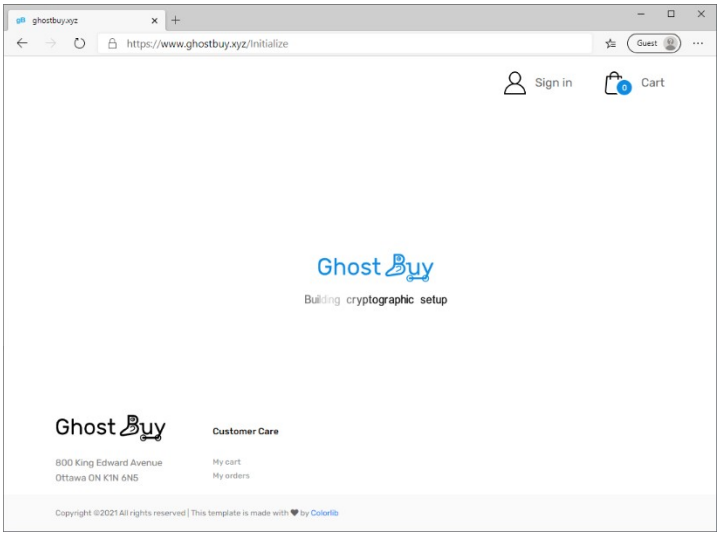

Figure 11: Initialization Page

When the customer starts browsing the *GhostBuy Prototype*, they are automatically redirected to the *Initialization Page*. The page shows a loading animation (see Figure 11) while the *Initial Cryptographic Setup* (see Section 5.4) is being built.

- ***Home Page*** *(https://www.ghostbuy.xyz/Home ➔Home.jsp)*

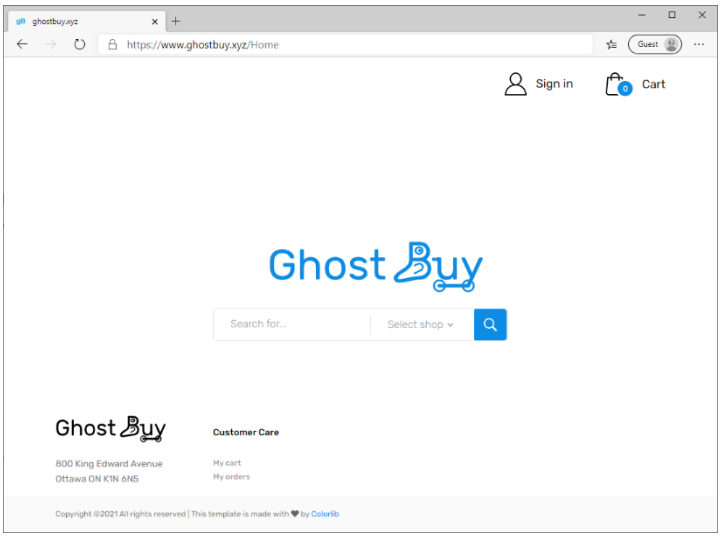

Figure 12: Home Page

When the *Initial Cryptographic Setup* has been built, the customer will be redirected to the *Home Page (*see Figure 12). Here, the customer enters the product search terms and selects which e-commerce website ("shop" for short) shall be searched (for details, see Section 5.6.2). When the first search is triggered the customer can choose to specify their personal details in order to check the search terms for – e.g. accidentally – contained personal information (for details, see Section 5.6.5).

- **Shop Page** *(https://www.ghostbuy.xyz/Shop?urlid=... →Shop.jsp)*

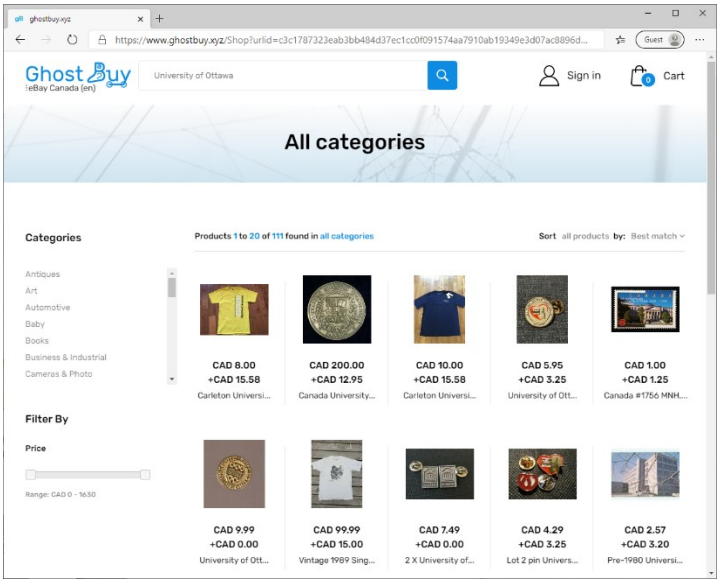

Figure 13: Shop Page

The search results are shown on the *Shop Page* (see Figure 13). The customer can refine the search by specifying more or different search terms, applying filters (product category, price range), changing the sorting order and navigating the paginated search results. When the customer clicks on a product on the *Shop Page,* they are taken to the *Product Page.*

- **Product Page** (https://www.ghostbuy.xyz/Product?urlid=... *→ProductDetails.jsp)*

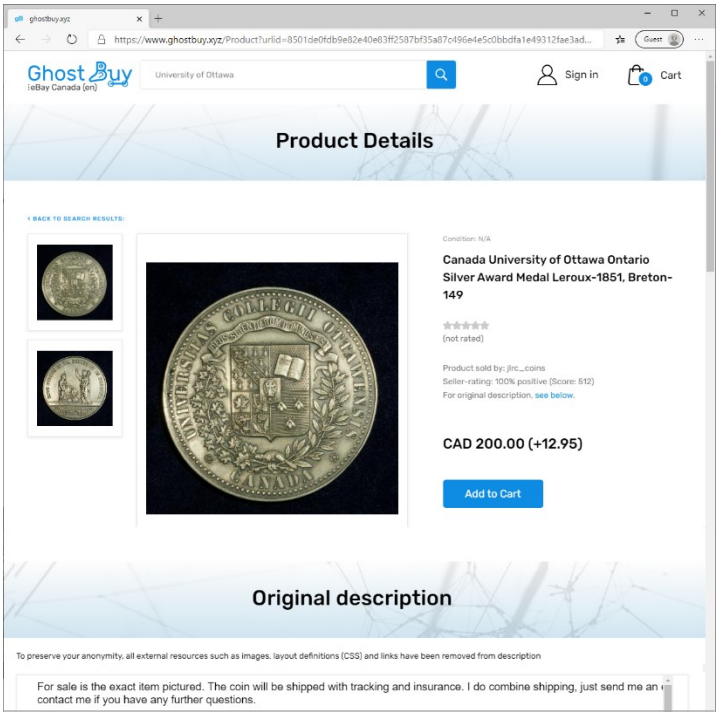

Figure 14: Product Page (composed screenshot)

The *Product Page* shows the details of a product such as images, condition, name, rating, seller name, seller rating, price and converted price if necessary (see Figure 14). The converted price is shown if the marketplace's currency is different from Canadian dollars since Canada is our assumed *GhostBuy Platform* companies' and customer location. In addition, the *Product Page* also shows the original description which is provided by the product seller and may be any valid HTML code. To protect the customer's anonymity, references to external resources are removed from the original description (for details, see Section 5.6.2). The

customer can choose to add a product to the shopping cart in which case they are taken to the *Shopping Cart Page*.

- **Shopping Cart Page** *(https://www.ghostbuy.xyz/Cart?urlid=... →ShoppingCart.jsp)*

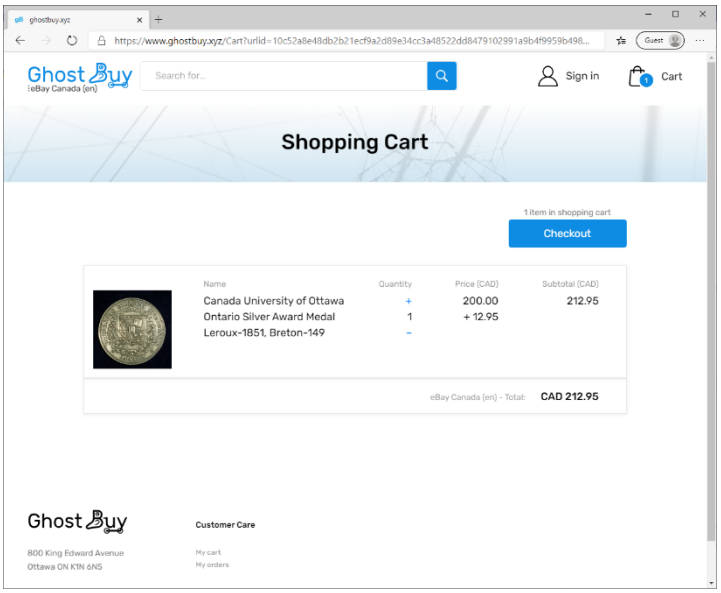

Figure 15: Shopping Cart Page

The *Shopping Cart Page* displays all the products the customer added to the shopping cart. In case the customer added products from several shops (e.g. eBay Canada and eBay United States) a separate shopping cart is shown for each shop. In the remainder of this thesis we will refer to any number of shopping carts with the term *Shopping Cart Collection*.[9] The customer can change the quantity of products, remove products from the *Shopping Cart Collection* and – by clicking it – open the *Product Details Page* for a product. If the customer clicks on the button to checkout the *Shopping Cart Collection*, they will be redirected to the *Sign-In Page* unless they already signed in before.

[9] The *Backend Application* manages all carts in a single encrypted object that we call *Shopping Cart Collection*.

- **Sign-In Page** *(https://www.ghostbuy.xyz/SignIn?urlid=...*[10] *→SignIn.jsp)*

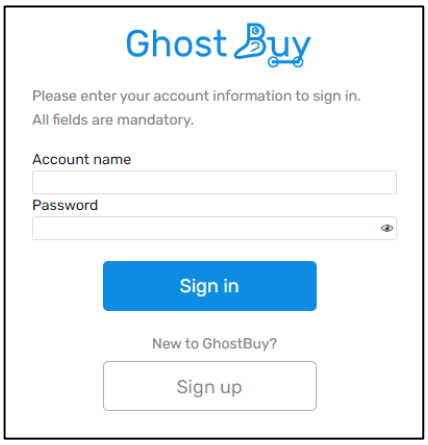

Figure 16: Sign-In Page (input mask)

On the *Sign-In* page the customer can choose to sign in by specifying their account name and password. Alternatively, the customer has the option to proceed to the *Sign-Up Page* to create a user account. If the customer signs in, they are taken to the appropriate target page afterwards (e.g., in the context of the browsing flow described here they are taken to the *Checkout Page*). This also works when the customer navigates backwards.[11]

- **Sign-Up Page** *(https://www.ghostbuy.xyz/SignUp?urlid=...*[12] *→SignUp.jsp)*

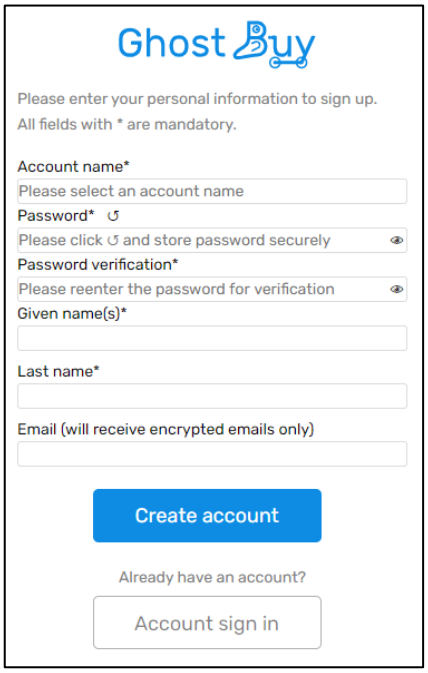

Figure 17: Sign-Up Page (input mask)

On the *Sign-Up* page the customer can choose to either create a user account or to go back to the *Sign-In Page*. To sign up, the customer must choose an account name, generate a password (and re-enter it correctly), specify their first and last name and optionally the email address (see Figure 17). See Section 5.6.7 for details on this. If the customer successfully signs up (i.e. creates a user account) they are taken to the appropriate target page afterwards (e.g., in the context of the browsing flow described here they are taken to the *Checkout Page*). This also works when the customer navigates backwards.[13]

[10] Here, the URL-ID is an optional parameter that usually is appended to the URL to be able to take the customer to the intended target page – or back the page on which the "Sign-In" button was clicked – after signing in.

[11] This is realized by requesting the authentication state via AJAX, thus complementing the authentication filter.

[12] See footnote 10.

[13] See footnote 11.

- **Checkout Page** *(https://www.ghostbuy.xyz/CheckOut?urlid=...*[14] *→CheckOut.jsp)*

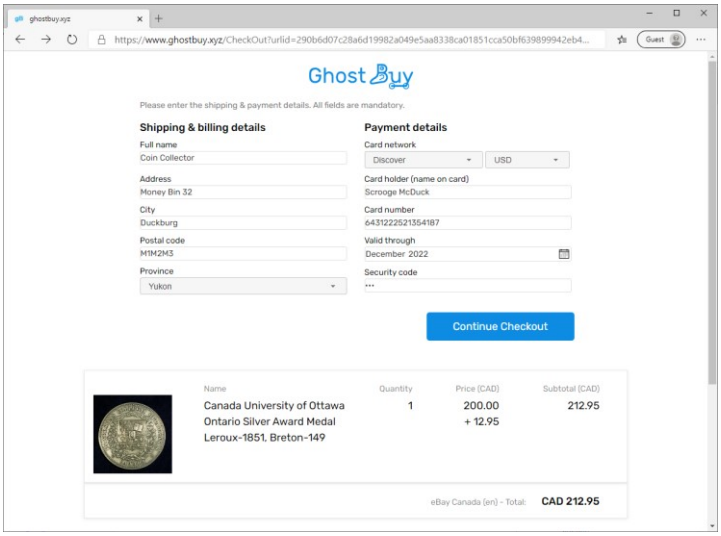

Figure 18: Checkout Page

The *Checkout Page* shows the same overview of the *Shopping Cart Collection* as the *Shopping Cart Page* but does not allow for changing product quantities or opening the *Product Details Page* (see Figure 18 – please note that the screenshot was edited to correct a typo). On the *Checkout Page* the customer must enter the shipping and payment information before being able to proceed with the checkout, i.e. go to the *Place Order Page*. This includes selecting the credit card currency which defines the *Shopping Cart Collection's* currency that is displayed later on the *Place Order Page.*

Please note: The complete checkout and payment process is only a simulation and does not involve real payments and – regarding the interaction with the respective e-commerce website (i.e. eBay marketplace) – no real ordering or shipping of products.

[14] The URL-ID is unused on the *Checkout Page* itself; since no customer-chosen parameters are pre-populated and the dynamic content that is loaded (shopping cart) does not depend on customer-chosen parameters, neither. It is however used to identify the *Checkout Page* as target page after signing in, where applicable. Therefore it is an optional part of the URL.

- **Place Order Page** *(https://www.ghostbuy.xyz/PlaceOrder →PlaceOrder.jsp)*

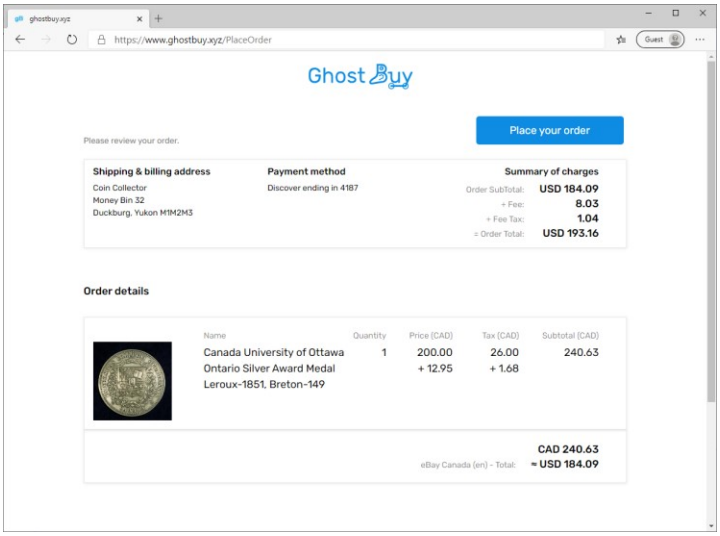

Figure 19: Place Order Page

The *Place Order Page* shows a summary of the previously entered shipping and payment details – with credit card information masked – and a summary of the calculated charges including fees & taxes, which in the prototype are based on hard-coded formulas. It also shows the *Shopping Cart Collection* again, but hereby includes the converted amounts for any shopping cart of which the currency is different from the credit card currency.[15] When the customer clicks the "Place your order"-button, the *GhostBuy Prototype* places the order (the ordering of products is only simulated in the *Backend Application*) and – depending on the result – takes the customer to the *Order Success Page* or *Order Failed Page*.

- **Order Success Page** *(https://www.ghostbuy.xyz/OrderSuccess →OrderSuccess.jsp)*

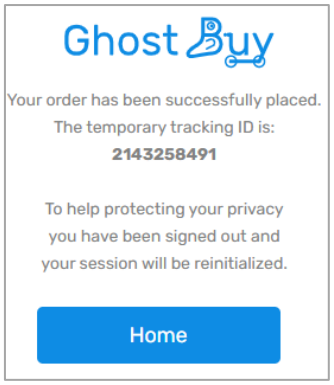

Figure 20: Order Success Page

When the order placement succeeds (all orders do, except when specifying the credit card security code "2020") the *Order Success Page* shows an according message that includes the *Frontend Tracking ID*. This ID is not a shipper's tracking ID. The *Frontend Application* assigns it to an order (for as long as the order is active) and for example it would be used to link a package that a customer returns to the related order at the *Backend Application*. The opposite but similar process is described in Section 5.7.

The *Order Success Page* provides a button to go back to the *Home Page*. Since the session is invalidated by the server after the successful order placement – which in effects logs the customer out – clicking this button will trigger the *Initial Cryptographic Setup*, again.

[15] The converted amounts are calculated based on fixed currency ratios.

- **Order Failed Page** *(https://www.ghostbuy.xyz/OrderFailed →OrderFailed.jsp)*

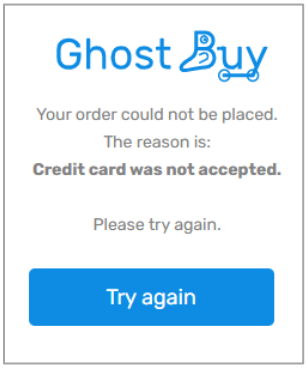

Figure 21: Order Failed Page

When the order placement fails (when specifying the credit card security code "2020") the *Order Failed Page* shows an according message and provides a button that takes the customer back to the *Checkout Page*.

Please note: The page header of the pages that are not part of the initialization process, authentication process or checkout process, contains direct links to the *Shopping Cart Page* and either the *Sign-In Page* (if not signed in) or the *Order History Page* and *Sign-Out Page* (if signed in). Also, the customer can trigger a product search on all these pages. Additionally, the page footer of all pages contains direct links to the *Order History Page* and the *Shopping Cart Page.* This enables alternative browsing flows, for example for returning customers.

- **Order History Page** (*https://www.ghostbuy.xyz/OrderHistory?ordernumber=... →OrderHistory.jsp*)

The *Order History Page* can be used to see the previous orders of a customer (see Figure 31 on page 78). To access it, the customer must be authenticated, respectively is asked to authenticate otherwise. The parameter "ordernumber" specifies the chronological number of the order the customer wants to see (e.g. 1 = first order of the customer). The order is retrieved in encrypted form and decrypted in the *Client Application* (see Section 5.6.9).

- **Sign-Out Page** (*https://www.ghostbuy.xyz/SignOut →SignedOut.jsp)*

The *Sign-Out Page* is shown when a customer signs out of the prototype. Opening the URL https://www.ghostbuy.xyz/SignOut, e.g. by clicking the Sign-Out button, triggers the sign-out. Client-side and server-side checks ensure in almost all cases[16] that the *Sign-Out Page* is

[16] The only case that we know of that is not considered, is when the *Sign-Out Page* is open, and the customer uses another browser tab to sign back in. This case could be handled by a client-side script, triggered by a counter.

not displayed when a customer is still signed in (e.g. when opening previous pages in the browser history). Depending on the context of the request/opening of the page, the request is alternatively redirected to an appropriate page, e.g. the *Sign-In Page* or the *Home Page*.

5.4. Overview of Cryptographic Keys Used in GhostBuy Prototype

As the following sections will show, multiple cryptographic keys are used within the *GhostBuy Prototype*. To avoid confusion, and to serve as a lookup reference, this section, respectively Table 3 and Table 4 give an overview of all custom cryptographic keys (i.e. excluding https-keys), even if they have not yet been introduced so far. For the reader's convenience, the keys can be identified with the *Key ID*, which is given in parentheses after each key in the following. The letters after the dash in each *Key ID* specify in which application the respective key is used (**C**lient, **P**ortal, **F**rontend, **B**ackend). However, the term and the *Key IDs* are only used within the text of this thesis, but not in the prototype implementation's source code and comments.

Table 3: Key usage by application (other than https)

	Portal	Client	Backend	Frontend
Portal	Portal Application Key (1-P)			
Client	Client Wrapping Key (2-P/C), *(forwarded: Client RSA Public Key (3-C/B))*	Client RSA Private Key (4-C), Client-side Secret Key (5-C)		
Backend	-	Client RSA Public Key (3-C/B), Backend Session Key (6-B/C)	Backend Application Web Service Key (7-B), Backend Application DAO Implementation Key (8-B)	
Frontend	-	-	-	Frontend Application Web Service Key (9-F), Frontend Application DAO Implementation Key (10-F)

Table 4: Overview of cryptographic keys in GhostBuy Prototype Implementation

Key ID	Application	Key Name	Key Generation Specification	Generated how and by	Storage	Usage
1-P	Portal	Portal Application Controller Key	AES, 256 Bit	Randomly generated once with Java keytool (manually)	Java keystore (Portal Application), imported upon application startup	Shared by all Portal Application Controllers (imported in parent class), used to encrypt and decrypt objects that are saved in session (e.g. Client Wrapping Key, Account DTO)
2-P/C	(Portal), Client	Client Wrapping Key	AES, 256 Bit	Randomly generated by **Portal Application**, once per session (custom Java class: AesGcmCrypto)	Session (Portal Application), encrypted with Portal Application Controller Key	Retrieved by Client Application (from Portal Application) on request to encrypt and decrypt session-related information (including other keys) stored in IndexedDB
3 -C/B	(Client), Backend	Client RSA Public Key	RSA, Modulus 4096 Bit, Exponent: 65537, Hash: SHA-512	Randomly generated as key pair by Client Application, once per session (custom JavaScript: rsa_oaep_functions)	IndexedDB (Client Application), encrypted with Client Wrapping Key	Sent to Backend Application (trough Portal Application) which herewith encrypts Backend Session Key to share it with Client Application (through Portal Application)
4-C	Client	Client RSA Private Key	see above	see above	IndexedDB (Client Application), encrypted with Client Wrapping Key	Used to decrypt Backend Session Key received from Backend Application (through Portal Application)
5-C	Client	Client-side Secret Key	AES, 256 Bit	Password-derivation by Client Application (custom JavaScript: aes_gcm_functions): PBKDF2, 32 Byte salt, 100,000 iterations	IndexedDB (Client Application), encrypted with Client Wrapping Key	Used to encrypt and decrypt Backend Session Key that is saved with successful order (Frontend Order DBO) to later decrypt respective order in Order History Page
6-B/C	Client	Backend Session Key	AES, 256 Bit	Randomly generated by **Backend Application**, once per session (custom Java class: AesGcmCrypto)	IndexedDB (Client Application), encrypted with Client Wrapping Key; later also saved in Frontend Order DBO in Frontend Application database, encrypted with Client-side Secret Key	Used to encrypt Client Application requests (parameters) and decrypt content from Backend Application (product information, images, shopping cart collection, shopping cart item count etc.)
see above	Backend	see above	see above	see above	"Virtual Session", i.e. session (Portal Application), but encrypted with Backend Application Web Service Key	Used to decrypt Client Application requests (parameters) and encrypt content for Client Application (product information, images, shopping cart collection)
7-B	Backend	Backend Application Web Service Key	AES, 256 Bit	Randomly generated once with Java keytool (manually)	Java keystore (Backend Application), imported upon application startup	Shared by all Backend Application Web Services (imported in parent Backend Webservice class), used to encrypt objects that are saved in "Virtual Session" at Portal Application (e.g. Shopping Cart, Backend Session Key) or in **Frontend Application** DB ("remote order ID")
8-B	Backend	Backend Application DAO Implementation Key	AES, 256 Bit	Randomly generated once with Java keytool (manually)	Java keystore (Backend Application), imported upon application startup	Shared by all Backend Application DAO Implementations (imported in parent Backend DAO Implementation class), used to encrypt objects that are saved in Backend Application database
9-F	Frontend	Frontend Application Web Service Key	AES, 256 Bit	Randomly generated once with Java keytool (manually)	Java keystore (Frontend Application), imported upon application startup	Shared by all Frontend Application Web Services (imported in parent Frontend Webservice class), used to encrypt objects that are saved in **Backend Application** DB ("remote order ID")
10-F	Frontend	Frontend Application DAO Implementation Key	AES, 256 Bit	Randomly generated once with Java keytool (manually)	Java keystore (Frontend Application), imported upon application startup	Shared by all Frontend Application DAO Implementations (imported in parent Frontend DAO Implementation class), used to encrypt objects that are saved in Frontend Application database

5.5. Initial Cryptographic Setup

We started the actual prototype implementation by programming the so-called *Initial Cryptographic Setup.* It is, hence the name, the initial setup process that establishes the cryptographic foundation for most subsequent information exchanges with and within the *GhostBuy Prototype* and it was therefore a technical prerequisite for all further implementation steps*:*

The *Initial Cryptographic Setup* is automatically initialised when a customer browses the *Portal Application's* domain (www.ghostbuy.xyz). Hereby a servlet filter (session filter) ensures that this only happens once as long as the customer's web session remains active. Essentially, the *Initial Cryptographic Setup* process creates and exchanges cryptographic keys between the *Portal Application* and the *Client Application, i.e. the Portal Application's client-side functionality* that is executed in the customer's web client (browser), and between the *Client Application* and the *Backend Application:*

- One symmetric key (AES256), that we call *Client Wrapping Key (2-P/C)*, is generated and provided by the *Portal Application* and used by the *Client Application* to encrypt content in the customer's browser-local *IndexedDB*.
- The public key of an asymmetric key pair (RSA4096), called *Client RSA Public Key* (*3-C/B*), is generated in the *Client Application* and sent to the *Backend Application* (through the *Portal Application*) which can therefore send encrypted information to the *Client Application.*
- Another symmetric key (AES256), called *Backend Session Key (6-B/C),* is generated by the *Backend Application* and sent to the *Client Application* (encrypted with the *Client RSA Public Key* (*3-C/B*)). This ensures that information can be exchanged between the *Backend Application* and the *Client Application* without the *Portal Application* and *Frontend Application* being able to read that information.

The *Initial Cryptographic Setup* also provides for the *Backend Application* to be able to create and handle a virtual shopping cart (collection) for each web session. This is despite the fact that the *Backend Application* is actually not aware of the web sessions as they are being managed by the *Portal Application.* This is achieved via a technique that we call *Virtual*

Session: Essentially, the approach of this technique is that the *Portal Application* saves an encrypted *Shopping Cart Collection* – that has been created, encrypted and provided by the *Backend Application* – to the web session. Whenever the *Backend Application* needs to update the *Shopping Cart Collection*, the *Portal Application* restores the encrypted *Shopping Cart Collection* from the web session and provides it to the *Backend Application* which can then decrypt and update it. Before the *Backend Application* returns the *Shopping Cart Collection* to the *Portal Application*, it re-encrypts it (for details please see Section 5.5)

Since the *Initial Cryptographic Setup* is the base for most of the further communication with and within the prototype, its understanding is crucial for comprehending the subsequent data processing and data flows (see Section 5.6). Therefore, in the following, the steps of the setup are described in more detail. For the sake of comprehensibility we hereby do not give all the details such as which exact component of an application performs a certain step and how each process step is triggered (a more detailed version of this process description can be found in Appendix D).

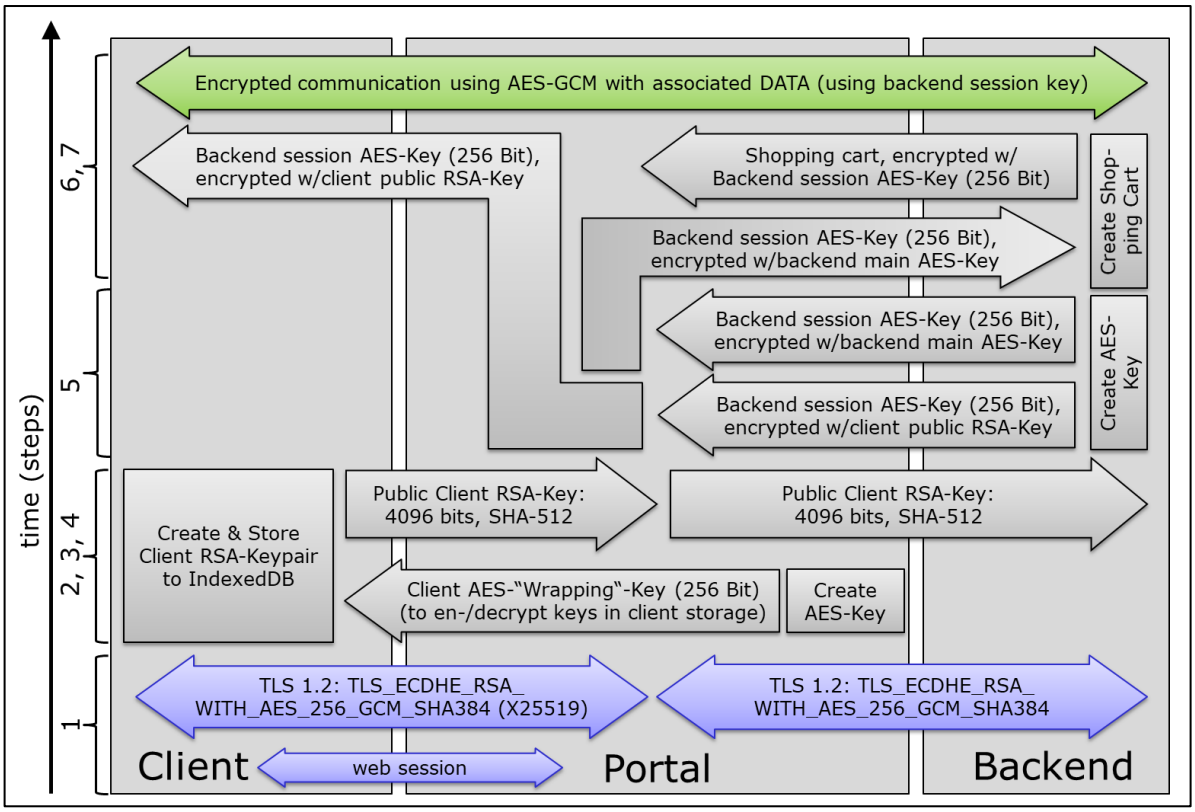

Figure 22: Initial Cryptographic Setup process

Figure 22 illustrates the process described in the following:

1. The first step of the initial setup process is triggered by a customer browsing the *Portal Application's* domain *(www.ghostbuy.xyz).* The customer's web browser establishes a secure connection to the *Portal Application's* web server who is configured to use TLSv1.2 with one of two recent, secure cipher suites[17]. It is important to note that the complete communication between the customer's web client (browser) and the *Portal Application* will take place over this secure connection, whereby – since the https protocol is stateless – each new request will use a new connection. The *Portal Application* creates a web session for the client. To identify the web session, it assigns a web session ID that will be recorded in the customer's browser by setting a session cookie (as part of the server response). That means that the *Client Application* can use the keys and other information that is part of the *Initial Cryptographic Setup* only as long as the web session remains active on the server side and the web client has access to the session cookie. Most tested browsers – with the exception of Opera – by default delete a session cookie when they are closed so that closing the browser effectively blocks any subsequent user from accessing/continuing the web session.

2. The *Portal Application* creates an AES256 key and stores it in the web session in encrypted form.[18] This key, that we name *Client Wrapping Key (2-P/C),* will be used for encrypting and decrypting sensitive session-related information that is stored in the customer's browser-local *IndexedDB*.

3. Next, the *Client Application* (in the prototype this is JavaScript executed in the browser[19]) creates an (asymmetric) RSA[20] key pair which it encrypts and saves to the

[17] TLS_ECDHE_ECDSA / RSA_WITH_AES_256_GCM_SHA384: These are the two most secure cipher suites at time of implementation of which at least one is supported by all tested browsers.

[18] For encryption it uses a *Portal Application*-specific AES256 key that we call *Portal Application Controller Key,* see *Key ID* 1-P in Table 4.

[19] The *Client Application* could be implemented in many ways, e.g. also as a native mobile device application, as long as the used protocols are compatible to the *Portal Application.*

[20] *RSA-OAEP* with 4096 bits, using SHA-512 as hash function

browser-local *IndexedDB*.[21] To encrypt it (and to later decrypt it), the *Client Application* uses the AES256 key that was created by the *Portal Application* in the previous step, hence the name *Client Wrapping Key (2-P/C)*. To obtain the *Client Wrapping Key (2-P/C)* from the *Portal Application*, the *Client Application* performs an AJAX request. Since this and other requests are implicitly authenticated via the web session ID, the *Client Wrapping Key (2-P/C)* can only be requested from within this browser and only as long as the web session is active. To recall, although the key is called *Client Wrapping Key (2-P/C)* it is also used to encrypt/decrypt all other sensitive session-related information in the browser-local *IndexedDB*.

4. After saving the RSA key pair, the *Client Application* sends its public part, the *Client RSA Public Key* (*3-C/B*), to the *Portal Application.* The *Portal Application encrypts and* saves the key in the web session.[22] Since the key is a public key, it is not strictly necessary to encrypt it, but it is intended to keep the session – which is stored in the server's working memory – unidentifiable to adversaries.

5. Now, the *Portal Application* requests a web service at the *Backend Application* to create another AES256 key. This key, that we named *Backend Session Key (6-B/C),* is returned in two separately encrypted forms. One form is encrypted with the client's *Client RSA Public Key* (*3-C/B*) (which was provided by the *Portal Application* as part of the request*).* The second form is encrypted with an AES256 key that is shared by all *Backend Application* web services and therefore called *Backend Application Web Service Key* (*7-B*). The *Portal Application* saves both encrypted key forms to the web session. Due to this, the *Backend Application* does not need to be aware of the different web sessions to "keep track" of the *Backend Session Keys (6-B/C)* that it created for these sessions: The *Portal Application* will provide the encrypted *Backend Session Key (6-B/C)*[23] with each subsequent (encrypted) request on behalf of a certain client. The *Backend Application* only needs to decrypt the key in order to be able to decrypt the

[21] It is saved as to separate keys: *Client RSA Public Key* (3-C/B) & *Client RSA Private Key* (*4-C*)

[22] Encrypted with *Portal Application Controller* (*1-P*).

[23] Encrypted with *Backend Application Web Service Key* (*7-B).*

request of the client. We call this technique *Virtual Session,* hence the name *"Backend Session Key" (6-B/C)*.

6. After the two forms of the *Backend Session Key (6-B/C)* have been saved, the *Portal Application* requests the *Backend Application* to create a new (empty) *Shopping Cart Collection*. As outlined in the previous step, for this it provides the encrypted *Backend Session Key (6-B/C)* [24] so that the *Backend Application* can use it to encrypt the *Shopping Cart Collection.* When the *Portal Application* receives the *Shopping Cart Collection* (that it cannot decrypt), it saves it to the web session (i.e. it becomes part of the *Virtual Session*).

7. Subsequently the *Client Application* obtains the RSA-encrypted *Backend Session Key (6-B/C)* [25] (that was saved to the web session in step 5) from the *Portal Application* and decrypts it using its private RSA key, the *Client RSA Private Key* (*4-C*), that it retrieves from the *IndexedDB*.[26] It re-encrypts the *Backend Session Key (6-B/C)* with the *Client Wrapping Key (2-P/C)* and saves it to the *IndexedDB*. Since the *Client Application* has now obtained all necessary information, the *Initial Cryptographic Setup* is completed, respectively *built.*

When the *Initial Cryptographic Setup* has been built, all further communication between the *Client Application* and the *Backend Application* is encrypted with the *Backend Session Key (6-B/C),* which means that the *Portal Application* (and the *Frontend Application*) cannot read the content of this communication although it de-facto defines the structure of the exchanged encrypted information: To recall, in the *GhostBuy Prototype* the *Client Application* executes JavaScript code that is being provided by the *Portal Application*.

[24] Encrypted with *Backend Application Web Service Key* (*Key ID 7-B).*

[25] Encrypted with *Client RSA Public Key* (*Key ID 3-C/B).*

[26] The *Client Application* retrieves the *Client RSA Private Key* (*Key ID 4-C*) from the *IndexedDB* and decrypts it with the *Client Wrapping Key (2-P/C)* that it also obtains from the *Portal Application.*

The complete process of building the *Initial Cryptographic Setup*, as described above, takes only about 2 to 5 seconds to complete[27] although it involves many different controllers, web filters, web services and requires multiple – transparent – web page forwards and redirects (for details, see Appendix D) as well as key generations, encryptions and decryptions. Meanwhile, the customer sees a loading animation that reads "Building cryptographic setup". The process is designed in a way that it can be resumed or resumes automatically in the unlikely case that e.g. the customer manually interrupts it or retriggers it before completion.

The process could presumably even be sped up slightly by restructuring the *Client Application's* initialization script and making minor adjustments to the initialization states recorded by the *Portal Application*. In general, this would be no bigger challenge. However, due to a lack of time and considering the comparably small advantages this change would likely yield, we did not implement it.

Finally, we would like to point out that we argue that the usage of a *Client Wrapping Key (2-P/C)* that is stored and provided by the *Portal Application* further mitigates the risk of so-called *Key Exchange API Attacks* on cryptographic keys used by the *Web Crypto API* (for details, see Section 6.3.5).

5.6. Subsequent Data Processing and Data Flows

The following two sections 5.6.1 and 5.6.2 describe the data processing and data flows that are necessary to display web page content that either is de-facto static, such as the general layout and placement of headers/footers, logos and content areas, or that is dynamic, but does not depend on parameters chosen by the customer as e.g. the list of selectable shops. Sections 5.6.3 to 5.6.6 will describe how search terms and other parameters are being processed and how the client state regarding the chosen parameters is being maintained. Finally, sections 5.6.7 to 5.6.9 will describe the customer authentication process, how order information from the *Frontend Application* and the *Backend Application* is being merged and how orders are authorised, placed, and prepared for review in the order history.

[27] Tested using current versions of the browsers Edge (non-legacy), Chrome, Firefox, Opera. A more detailed performance analysis is given in Section 6.4.3.

5.6.1. Loading Web Pages with De-Facto Static Content

Please note that the following assumes that the *Initial Cryptographic Setup* has been completed although most parts of the description in this section also apply to the *Initial Cryptographic Setup* itself.

When the customer browses any URL of the *GhostBuy Prototype* (i.e. any URL of the *Portal Application,* e.g. *https://www.ghostbuy.xyz/Home*) the browser submits an https-get request to the *Portal Application.* The request is being handled by the corresponding controller (web servlet)[28] who maps the requested URL to a web page. Hereby all pages that load non-static content are being served by a dedicated controller.[29] The reason for that was mainly to keep things structured during the development but, in addition, some controllers also implement page specific functionality to process user inputs. Also, dedicated controllers should scale better under heavy work loads but a real-world implementation should also process requests asynchronously.

The source files of the web pages that are served by the controllers are *Jakarta Server Pages (JSP).* They define the pages' layouts using HTML5 markup including the use of *Cascading Style Sheets (CSS)*. The actual web pages (i.e. the pages to which the controllers map the requests), such as the *Home Page* (Home.jsp), include other JSP files that e.g. define the page footer (HTMLFooter.jsp). This is to avoid markup/layout redundancies and inconsistencies.

However, although JSP supports several mechanisms as for example JSP scriptlets and JSP expression language to display dynamic content, most web pages served by the *Portal Application*'s controllers do not use these mechanisms. When those web pages are loaded by the web client (browser) their content is "at first" de-facto static.[30] The reason for this is that

---

[28] In some cases, a servlet filter previously redirects the request to another page (e.g. when the customer has to sign-in to access a certain web page).

[29] There are three static pages to show either an application error (for example if we forgot to start the Backend Application), a page-not-found error (also known as error 404, shown when a non-existing URL is being requested) or the session expired message. These pages are being served by a single controller that accepts an according parameter to display the respectively requested page.

[30] The only exceptions to this are the pages showing an order success or failure in which we used JSP expression language to display textual information (Order ID or error message) that is being added to the request object.

most of the dynamic content that we need to display on the web pages, such as product information, may only be known to the *Backend Application* and therefore has to be added *after* the page has been served by a controller of the *Portal Application,* i.e. after it has been loaded by the web client (browser). In fact, the content *could* theoretically be inserted – e.g. in encrypted form in hidden fields – into the page by the *Portal Application's* controllers *before* being served to the browser. Then, this content would be decrypted and displayed on the page e.g. by running JavaScript code. However, the downside of that approach would be that the responsiveness of the web pages would be reduced as for example the time between clicking a link and seeing the target page loading would increase. This is due to the fact that the sole processing and provisioning of the JSP page itself can naturally be done way faster without additionally requesting content from the *Backend Application.* This is true especially if serving this request would involve an API call to the respective e-commerce website (e.g. eBay), e.g. to obtain product information.[31] Providing the encrypted content directly within the web pages served to the browser also would have another downside: The content would be cached in the browser which would give an offline attacker at least more encrypted content that theoretically could be decrypted if an attack was successful or even help to mount such an attack.

Due to this, the web pages with dynamic content that *GhostBuy's* customers see and interact with, which represent the user interface of the *Client Application,* are being composed by loading web pages with de-facto static content to define the structure and layout of the web page[32] and afterwards loading and adding the dynamic content. This is achieved by executing JavaScript code after the web pages have been loaded by the customer's web browser. The process is described in the next section.

---

[31] E.g. when the shopping cart shall be displayed (without adding a new product) an API call to the e-commerce website is not necessary.

[32] The layout can differ from device to device since the OneTech template includes two *Cascaded Style Sheets* for most pages, one for desktop browsers and one for mobile device browsers. During implementation, however, we focused on adapting the desktop browser layout and paid less attention to the mobile device browser layout. Although the page layout on mobile devices is therefore "slightly broken", we at least ensured that mobile devices could in general be used to browse and use the platform.

5.6.2. Dynamically Adding Encrypted Content

A "simple" example for dynamically adding encrypted content is the population of the shop selector drop-down box on the *Portal Application's* home page (see Figure 23). It is "simple" because it does not require sending customer-chosen parameters to the Backend Application. For the sake of comprehensibility we again do not give all the details such as which exact component of an application performs a certain step and how each process step is triggered (a more detailed version of this process description can be found in Appendix E). Figure 24 illustrates the process steps described in the following, with the numbers of the dataflow arrows corresponding with the numbers below:

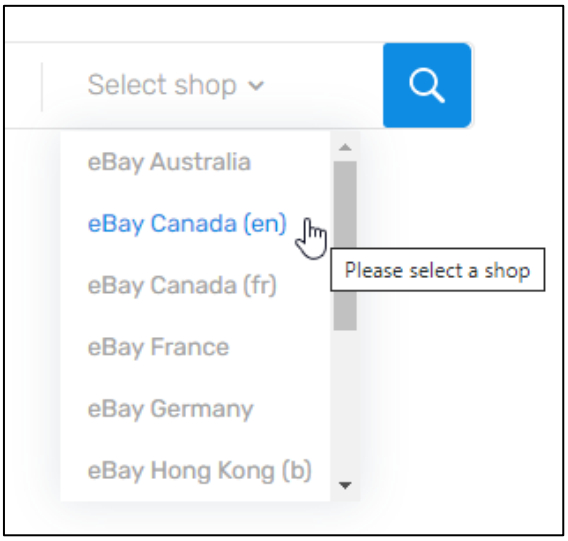

Figure 23: Shop selector drop-down box (populated)

1 & 2: When the home page has been loaded in the *Client Application*, i.e. by the customer's web client (browser), a web-page specific script is being executed that – via invocation of functions that are implemented in a custom JavaScript module – makes an AJAX request (*https-get*) to obtain the shop list from the *Portal Application*.

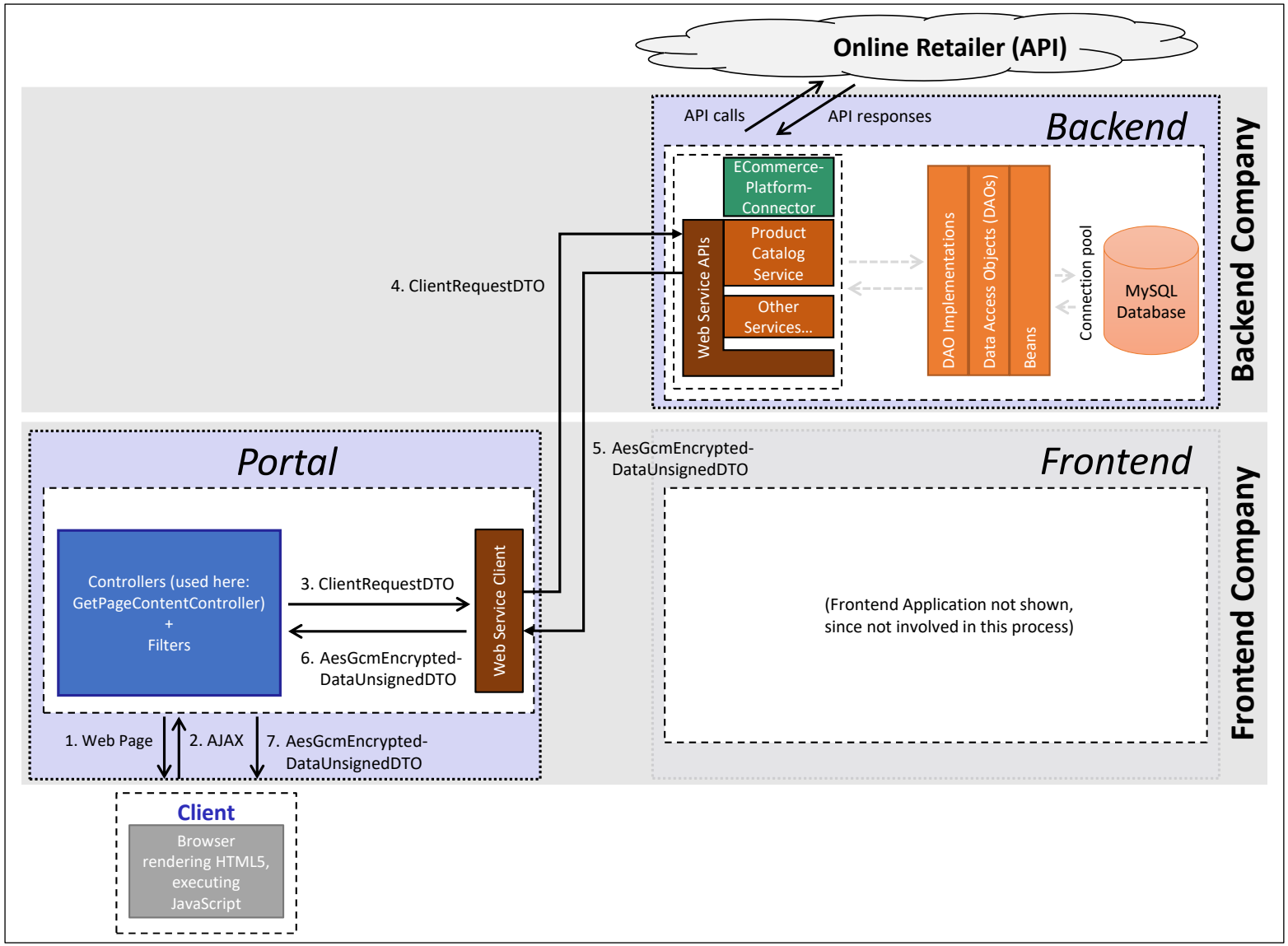

Figure 24: Page content request from Client Application to Backend Application

3 & 4: The *Portal Application* requests a web service at the *Backend Application* to provide the shop list. To enable the *Backend Application's* web service to encrypt the shop list with the client-specific *Backend Session Key (6-B/C),* the *Portal Application* passes the encrypted key[33] – encapsuled in a so-called *ClientRequestDTO* – to the web service.[34] To do so it uses a custom web service client that supports all methods being implemented by the *Backend Application's* web services*:* I.e. the web service client performs the web service request (providing the *ClientRequestDTO*), interprets the returned status code(s) and converts the received encrypted shop list (see next step) to an encrypted java object.[35]

5, 6, 7: The web service at the *Backend Application* assembled the shop list during startup.[36] It extracts the encrypted *Backend Session Key (6-B/C)* from the *ClientRequestDTO,* decrypts it, and uses it to encrypt the shop list which it returns as a custom, encrypted DTO[37] to the *Portal Application* via the web service client. The *Portal Application* – who is not able to decrypt the content of the DTO – returns it to the calling function in the *Client Application,* – i.e. the web page's JavaScript execution – with the *Client Application* receiving a JSON representation of the DTO.

Finally, the *Client Application* decrypts the content – for which it has to restore and decrypt the *Backend Session Key (6-B/C)* from the browser-local *IndexedDB* – using the *Web Crypto API*. It then manipulates the web page's *Document Object Model (DOM)* to populate the shop

---

[33] Encrypted with *Backend Application Web Service Key* (*7-B).*

[34] See step 5 of the *Initial Cryptographic Setup* described in Section 5.5.

[35] The web service itself returns a textual representation of the encrypted object using *JavaScript Object Notation (JSON)*

[36] The shop list is pre-generated during startup of the *ProductCatalogService* via iterating through all so-called *ECommercePlatformConnectors*. For the prototype we only implemented one *ECommercePlatformConnector,* namely the *EbayConnector.* Since eBay's shops (marketplaces) do not consistently support all API functions and we have not found an API with which they can be retrieved dynamically, this connector has the supported shops hard-coded into a static initialization function

[37] *AesGcmEncryptedDataUnsignedDTO*

selector drop-down box with the list. Please see Appendix M for an illustration of the main steps of the decryption and conversion process as performed in the *Client Application*.

Admittedly, it is not absolutely necessary to encrypt the shop list since the complete list itself is not secret (but the customer's shop selection is, see Section 5.6.3). However, we decided to encrypt it since we use an analogous process for other – more sensitive – content being provided by the backend. As stated above, this process description is to be seen as a simple example for all these similar processes. In addition, the shop list could later be customized based on a pre-selection made by the customer (e.g. "stores selling jewelry…") which would render it sensitive information, too.

Regarding privacy and anonymity, it should be noted that the *Portal Application* is aware of what *type* of content has been requested from the *Backend Application* (in this case the shop list), but it is never aware of the request parameters' content (e.g. selected shop, search terms, selected categories) and of course the returned content itself. This property provides the required level of anonymity and privacy:

It is obvious and no secret that a customer would use *GhostBuy* to search for or buy *something,* i.e. an *unknown product*. Therefore, the platform's goal is that the exact request parameters and the product stays unknown to the *Portal Application (*and the *Frontend Application*), hereby preserving privacy*,* and that the customer's identity stays unknown to the *Backend Application,* hereby preserving anonymity.

5.6.3. Client-Side Parameter Encryption and Submission

The example in the previous section where the encrypted shop list was requested and loaded into the web page described how content is dynamically added in the *Client Application* without disclosing it to the *Portal Application.* As mentioned therein, requesting the shop list does not require sending customer-chosen parameters from the *Client Application* to the *Backend Application.* However, the *GhostBuy Prototype* must of course implement such an option so that the customer can search for something, e.g. by specifying search terms and applying filters such as *product category* and *price range*. The *Client Application* also needs to pass on parameters to make the user experience customizable, e.g. by selecting the pagination size, i.e. the number of products shown on one page. Although some parameters

are less sensitive than others, none of the parameters may be revealed to the *Portal Application* (and *Frontend Application*), just like the dynamically loaded content itself.

We achieve this by encrypting search terms and other parameters in the *Client Application* using the *Backend Session Key (6-B/C)*. In the prototype, the encryption in the *Client Application* is performed by JavaScript functions that utilize the *Web Crypto API*. Apart from the parameter encryption, the remainder of the process is almost identical to the example from the previous section: The only two differences are, firstly, that the content is requested using an AJAX request that uses *https-post* instead of *https-get.* This allows the encrypted parameters to be sent to the *Portal Application*[38] in the https request body – the reasons to do so are explained in Section 5.6.6. Secondly, the *Portal Application* adds the parameters to the *ClientRequestDTO* so that the web service at the *Backend Application* can decrypt and use them. Additionally, the *Backend Application's* web service uses a custom e-commerce platform connector[39] to obtain the requested product information from the respective e-commerce website. In the prototype we implemented only the connector that is necessary to obtain information from several eBay marketplaces.[40]

5.6.4. Special Processing of Image Data and Original Product Description

Although the previous sections have described the main concepts for requesting and adding dynamic content such as the shop list, product descriptions and others we would like to point out two types of content that require special processing, especially by the *Backend Application.* Although the problematic content was specific to the eBay API's way of content provision – and therefore is handled within our custom *EbayConnector* – one can assume that similar problems would arise with similar content from other e-commerce platforms and would have to be processed accordingly:

The first problematic type of content are the product images which for example are shown on the *Shop Page* and *Product Page*. In the eBay API, product images are provided as source links like https://i.ebayimg.com/images/g/<ImageID>/s-l1600.jpg. The problem with this is

[38] A description that details the respective components can be found in Appendix F.

[39] Class "*ECommercePlatformConnector"*

[40] Class "*EbayConnector"*

that firstly the *Client Application* may not access GhostBuy-external sources because of anonymity and privacy reasons. This means that we cannot simply add these source URLs in the same way as we dynamically add other encrypted content, since the *Client Application*, i.e. the browser, would load these images from the respective URLs. Secondly, we also cannot provide these images via source URLs e.g. by encrypting them and making them available at the *Portal Application* since the browser cannot – without further ado – load encrypted images via source URLs.

Therefore, the key to our solution for dynamically adding encrypted product images is that product images are downloaded by the *Backend Application* and converted to data URLs. Thus, they can be encrypted with the *Backend Session Key (6-B/C)* and provided like the other product information, too. In the *Client Application* the encrypted data URLs are decrypted and inserted into the web page DOM. To avoid bottlenecks, respectively processing delays due to sequential downloading and conversion of the images, this part of the content provisioning is parallelized within the *Backend Application*. As our prototype shows, this approach works well and reasonably efficient (see Section 6.4.3), but it should be noted that the conversion to encrypted data URLs – which are base64-encoded – results in data overhead. Alternative approaches, in which for example images are transmitted as encrypted BLOBs instead of encrypted base64-encoded data URLs, might therefore be even more efficient. However, our approach mainly served to show that the problem can in principle be solved efficiently.

The second problematic type of content is the "*Original Description"* that is provided by the respective sellers of a product on eBay: This description is html code, which on the original eBay marketplaces is embedded into an iframe and can also include embedded and external resources such as images, *Cascaded Style Sheets* and scripts. Since the *Original Description* often contains important information about the product, we sought to include it in the *Product Page*, too. In general, this is no problem since the *Original Description* html code can be retrieved via the eBay API as well. However, since the code can contain images and other embedded or external resources, including it into the *Product Page* results in the same problems as for the product images: The *Client Application* may not access GhostBuy-external sources for anonymity and privacy reasons and the content cannot simply be provided via source URLs when we would download and encrypt it in the *Backend Application*. Moreover,

the content is even more problematic since it is much less standardized than the product images, which means that for example the included external sources vary greatly. Therefore, we decided not to try to convert these external resources to internal resources as we did with the product images. Instead, we sanitize the content by stripping it from all external resources at the *Backend Application* before returning it, encrypted, to the *Client Application*. Of course, this solution has the disadvantage that the *Original Description* often does not look like the seller wanted it to look. Yet, our approach is meant to show that we are aware of this problem rather than actually providing a complete solution to it.

Therefore, a real-world implementation might seek to implement a less efficient but nicer solution, for example by loading and rendering the *Original Description* in some kind of sandbox environment, capturing the visual representation as a screenshot and embedding this as an encrypted image into the web page (as a data URL). Since this approach, however, would only work well for static content and would also result in potentially very large, encrypted image resources (data URLs) being transferred, one might alternatively seek to parse the html code thoroughly and attempt to convert the external resources into internal resources.

5.6.5. Client-Side Search Term Evaluation

The concept of *Separation of Data* implies that the *Backend Application* only receives parameters from the customer that do not reveal the customer's identity. In particular, the *Backend Application* may not receive personal information such as the customer's name or address. At the same time, the customer-chosen parameters such as the search terms must be encrypted in the *Client Application* – in order to hide them from the *Portal Application* and *Frontend Application* (described in Section 5.6.3). This leads to the following problem: How can we ensure that the customer does not send personal identifiable information to the *Backend Application* when the *Portal Application* can indeed define the structure of the data (it provides the files that implement the *Client Application*) but may not check the plaintext parameters? We considered the following approaches to address the problem:

1. The *Client Application* could use public key cryptography to encrypt the customer-chosen parameters that it wants to send to the *Backend Application*. The *Portal Application* would use the public key from a key pair, which would be provided by the Backend Application, to pre-encrypt the customer's personal information. When the

Client Application sends the parameters – also encrypted with the Backend Application's public key – to the *Backend Application,* it would actually send them to the *Portal Application* which forwards them to the *Backend Application.* However, if the *Portal Application* would detect a match with the pre-encrypted customer`s personal information it would not forward the information but e.g., at least, asks for the customer`s consent to do so.

This approach has at least two downsides which ultimately made it impossible to use: Firstly, the personal identifiable information might be only a part of the customer chosen parameters or might vary otherwise. E.g. a user named "John Doe" might search for "John Doe computer", "John do" or "john doe". Therefore, detecting matches with pre-encrypted ciphertexts would require to pre-encrypt all potential variations; which is simply impossible or at least inefficient, since – especially regarding combinations with other words – the number of variations is very large. In addition, public key cryptography is less efficient than symmetric key cryptography.
Secondly, practical public key cryptography implementations such as *RSA-OAEP*[41] use padding schemes (*OAEP = Optimal Asymmetric Encryption Padding*) that prevent two independently computed encryptions of the same plaintext to yield the same ciphertext, which in most other use cases is a required property. In addition, the padding is randomized and does not accept an input equivalent to the *Initialisation Vectors* used in (symmetric) block ciphers. I.e. the padding cannot be controlled in order to have the encryption yield the same ciphertexts for the same plaintexts – although we would like to achieve exactly that. It is possible that other RSA implementations grant control of the padding, but the *Web Crypto API* does not do this. For these two reasons – the padding itself and that it is not controllable – the approach to match asymmetrically encrypted ciphertexts with precomputed asymmetric encryptions of known plaintexts is not realizable.

2. The second approach is similar to the first approach in that it would also be based on the *Portal Application* detecting matches in computationally altered representations of

[41] Which is also available in the *Web Crypto API*.

the plaintext parameters, from which, however, it is not possible to recover the original plaintext parameters. Unlike the first approach, however, this time the to-be-matched parameters would be hashed. The parameter hashes would be computed by the *Client Application* and sent to the *Portal Application* (along with the actual encrypted parameters). The *Portal Application* would match the hashes with pre-computed hashes of the customer's plaintext personal information.

The obvious downside of that approach is that, like in the first approach, the number of possible variations in the customer-chosen parameters is too high to efficiently pre-compute and match their hashes.

It is possible that this downside could be overcome by using a different kind of one-way function – instead of plain hashes – that would yield the same result for similar variations of a plaintext. One possible option for that is the so-called *fuzzy hashing* which has been researched e.g. by Kornblum [54] as well as Breitinger and Baier [55]. Yet, it stands to question if *fuzzy hashing* would be able to detect similarities in parameters when the identical part of the plaintext variation is small, e.g. smaller than the deviating part (e.g. "John Doe" and "John Doe high performance computer"). Yet, this could be researched in future work.

3. The third and actually implemented approach is the following: The *Client Application* simply checks the customer-chosen parameters itself before encrypting them and sending them to the *Backend Application* (via the *Portal Application*). Thus, it can detect partial matches with personal identifiable information while not revealing the plaintext parameters to the *Portal Application* and *Frontend Application*. One small downside of the approach is that the *Client Application* might be limited regarding its computational power, but we argue that the computational power necessary to perform the matching is likely smaller than the computational power that is necessary to perform the different types of encryption (and hashing) which is also performed in the *Client Application.* I.e. a client that is capable to run the more demanding parts of the *Client Application* in reasonable time will also be capable to perform the matching. This is the case even if the matching function was more sophisticated than the rather simple function that we implemented in the prototype. The second small downside is

somehow similar to the first in that it is about the client's capabilities: The client – at least in the prototype - needs to run JavaScript to perform the matching but restrictive browser security settings might not allow to do so. Yet, if the browser is not capable to use JavaScript, it cannot run the prototype implementation`s *Client Application* at all.

For the protype we implemented the above third approach to realize the following process:

1. When a customer triggers a product search – i.e. enters search terms and clicks the search button – for the first time (after completion of the *Initial Cryptographic Setup*) a popup dialogue asks if they would like to specify their personal details to check the search terms for potentially contained personal information (see Figure 25).[42]

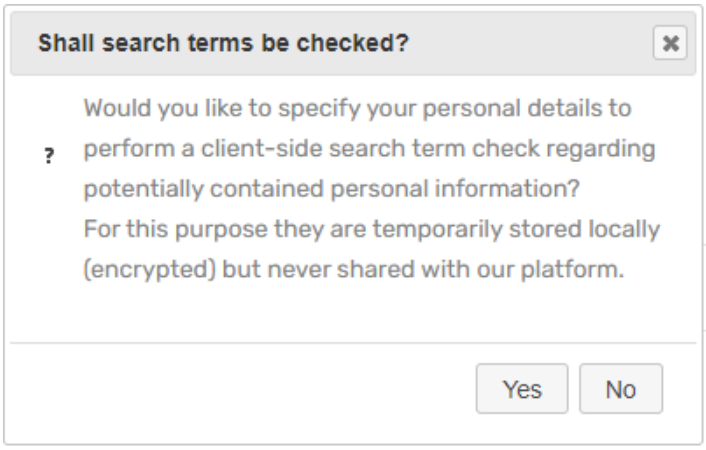

Figure 25: Search term check confirmation

2. If the customer agrees, a second dialog opens where the personal details can be entered. The input will be checked to comply with certain patterns (e.g. the postal code) and, if everything is OK, it will be encrypted with the *Client Wrapping Key (2-P/C)* – that the *Client Application* obtains from the *Portal Application* – and stored in the browser-local *IndexedDB*. Thus, the information can only be decrypted until the web session expires or the browser is closed (and the web session cookie is deleted).

3. Then, the Client Application checks the search terms for matches – including partial matches – with the personal details. For this it uses a custom scoring system that e.g. considers the sensitivity of personal information (e.g. we defined that the province is a less sensitive information than the postal code) and the extent of matches (partial,

[42] In addition, the customer is reassured that the personal information is encrypted, stored locally and never shared with the *GhostBuy Platform*.

multiple, complete match). This check and the next step will be repeated for each individual search triggered by the customer:

4. If it detects a match, the *Client Application* will inform the customer about this, quantifying the likelihood that personal identifiable information is contained (see Figure 26). The customer can decide to review the search terms or trigger the search anyways. If no match is detected the search is performed right away.

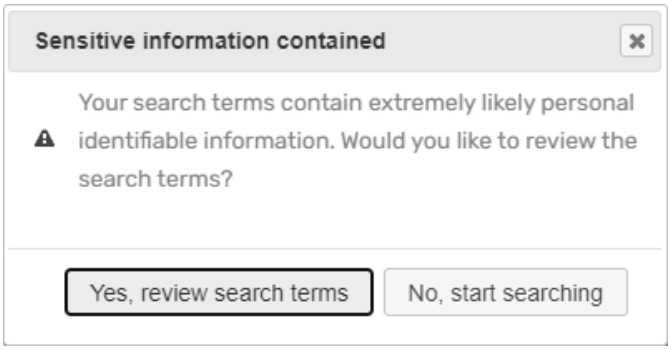

Figure 26: Sensitive information contained warning

If in step 1 the customer decides not to specify personal details, the search is performed right away, too. As with the personal details in step 2, the customer's preference is encrypted and saved in the *IndexedDB* so that the same applies for subsequent searches without the customer being asked again.

5.6.6. Client State Preservation During Navigation

Usually, web pages use URL parameters to submit customer-chosen parameters. As the name says, these parameters are submitted as part of the web URL. E.g. the URL https://www.amazon.ca/s?k=AMD+CPU&i=electronics contains the parameters *k = "AMD+CPU"* and *i = "electronics"*. The parameters are received and interpreted by the server of the web application, which returns the requested content and page layout (e.g. regarding pagination size), if applicable. When a user refreshes the web page or uses the browser history to open a previous page, the parameters are used to rebuild the page as it was before, although the returned content itself might have changed, e.g. due to new products being available.

As discussed in sections 5.6.3 and 5.6.5, in the *GhostBuy Architecture/Prototype* customer-chosen parameters such as search terms, product filters and pagination size must be encrypted in the *Client Application* before they are sent/forwarded to the *Backend Application*. This requires addressing the following problem:

Even a few customer-chosen parameters already expand to a large textual representation when they are encrypted. To illustrate this: A test showed that a small set of parameters expands into a string about 1000 characters long when encrypted and URL-encoded. This is about half of the overall URL length being supported by Microsoft Internet Explorer.[43] Although Internet Explorer is not supported by our prototype, this fact and also considering the not-so-nice "URL-aesthetics" (e.g. when copying & pasting an URL) we decided against using encrypted URL parameters. Instead, the encrypted parameters are sent in the request body of AJAX https-post requests.[44]

Yet, having to request dynamic encrypted content (using AJAX-calls) after a de-facto-static page loaded (see Section 5.6.2) but at the same time being unable – respectively unwilling – to use encrypted URL parameters leads to the next problem: How can we preserve knowledge about the chosen parameters while navigating from the page where the parameters were chosen to the page where the respective content shall be shown – i.e. while a page load happens? For example, if a customer enters a specific search term on the *Home Page* and then clicks on the search button to have the results of the search displayed on the *Shop Page*, how does the *Shop Page* – after having been loaded – "know" which encrypted search terms and other parameters to send to the *Backend Application*? Our solution to this is that the parameters are being encrypted and stored in the browser-local *IndexedDB* before the target page – in this case the *Shop Page* – is loaded. Once the page is loaded, the parameters are restored from the *IndexedDB* and sent to the *Backend Application* (via a *Portal Application* controller).

This process works without issues, if a customer sequentially performs search after search (e.g. refining the search terms). In reality, though, this often may not be the case, since e.g. a customer might use the browser's history to jump to a certain page or the customer might open several tabs in the browser – which therefore all run in the same session – for example to compare the results of multiple searches. If the customer refreshes any of the pages or jumps to a certain page, the *Client Application* would always retrieve the last parameters that have

[43] https://support.microsoft.com/en-ca/help/208427/maximum-url-length-is-2-083-characters-in-internet-explorer

[44] At first glance this seems to contradict the general usage of the *https-get* and *https-post.* Therefore in Appendix G we briefly discuss why we think it does not.

been encrypted and saved to the *IndexedDB*, at least if we store only a single set of encrypted parameters in the *IndexedDB*.

The obvious solution to this problem is to distinguish between multiple sets of encrypted parameters. To do so, each set is assigned an identifier which is added to the URL as an URL-parameter that we call *URL-ID*. This means that the URL – including the *URL-ID* – defines, which encrypted parameter set is restored from the *IndexedDB* and thus which content is requested and loaded into the page. Using an identifier has the advantage that it – if chosen carefully – does not contain sensitive information but at the same time can be much smaller than the encrypted parameter sets themselves. Regarding the identifier we decided against using a random number or a counter, instead we generate the identifier by hashing the parameter set – before encryption – with the SHA-256[45] hashing algorithm. This ensures that a certain parameter set will always yield the same identifier, e.g. when a customer performs the same search twice, which therefore avoids storing redundant encrypted parameter sets. However, to avoid that same sets of parameters yield the same identifiers across different sessions (at different times, browsers, devices etc.), a session-specific secret 256 bit salt is added during hashing. This ensures that the identifier does not allow any conclusions to be drawn about the parameter set and that URLs cannot be shared or otherwise transferred between different sessions, neither on purpose nor by accident – and also that 256-bit collision resistance is provided.

Our above solution for preserving the client state – regarding selected parameters – between page loads and to allow for browser (history) navigation is described in more detail in Appendix I – where the transition between the *Home Page* and the *Shop Page* should be seen as an example for all transitions. Additionally, Figure 27 illustrates the steps described therein.

[45] I.e. Secure Hash Algorithm 2 (SHA-2) with 256 bits output.

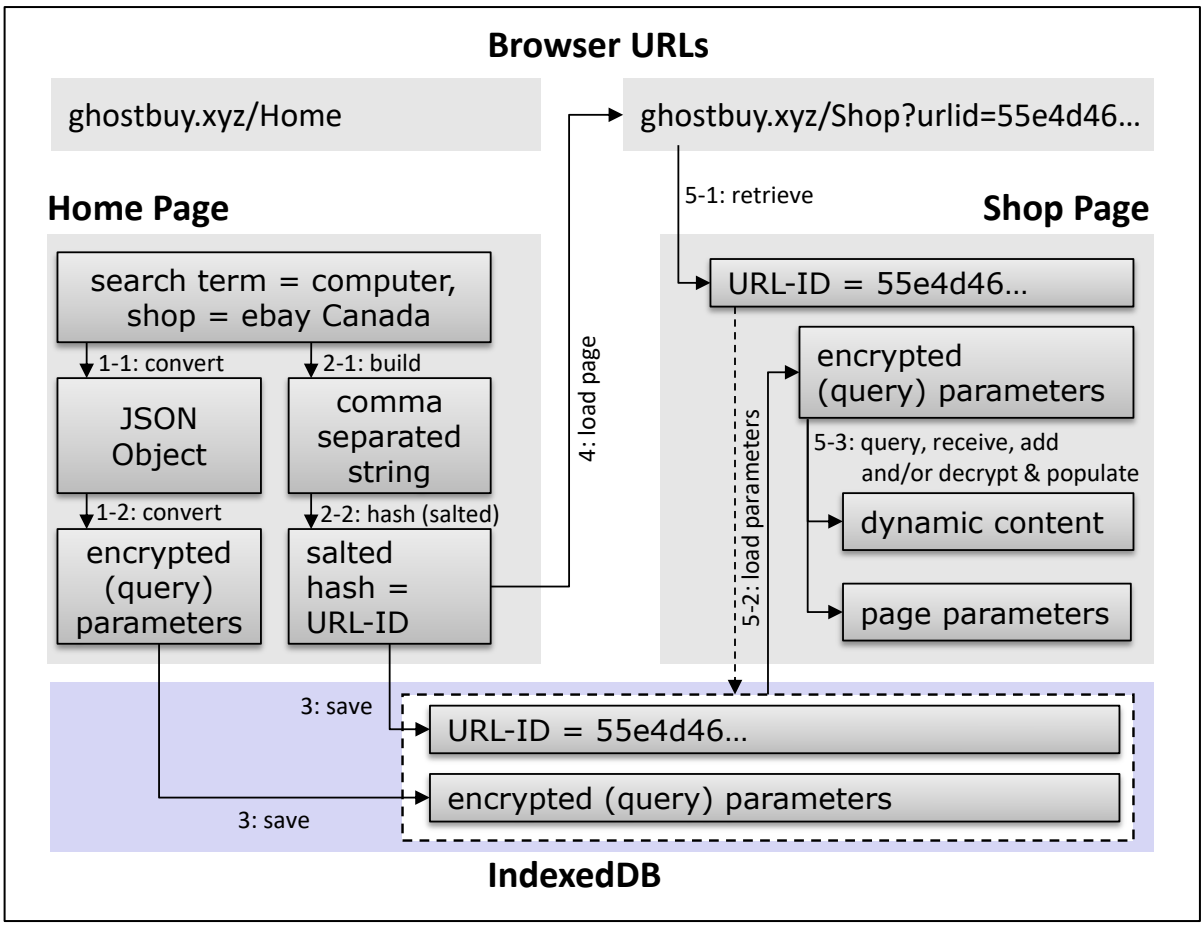

Figure 27: Client state preservation during navigation

Please note:

- If a *URL-ID* cannot be found in the *IndexedDB* (e.g. when trying to copy & paste a URL between two different sessions), the *Client Application* will either forward to the portal's home page or – if applicable – load the page without populating the page parameters, i.e. it will show the default parameter selection (e.g. on the *Home Page* no shop would be pre-selected in the drop-down box).
- The *URL-ID* is also used to identify the target page to which a user shall be redirected after logging in, i.e. either the page where the user clicked the "Sign-In"-button or the page that the user tried to open but which required authentication.
- On some pages the *URL-ID* of the previous page is appended to the URL as so-called *Back-URL-ID*. By introducing this parameter, we allowed for navigating "back"[46] to

[46] Technically, this actually is a forward navigation.

the previous page by clicking a link in the page. For example, one can return to the search results on the *Shop Page* by clicking a "backlink" on the *Product Page*.

5.6.7. Client to Frontend Authentication (Customer Authentication)

When a customer wants to checkout the *Shopping Cart Collection* or wants to see previous orders, they are required to sign-in to *GhostBuy*, which means they have to authenticate themselves. If the customer does not yet have a user account, they must "sign up", i.e. they have to create a user account. After signing up, the customer is authenticated, too – at least for as long as the session remains active.

In general, our authentication process is not different from other password-based authentication processes, in the sense that the customer must choose an account name and a password when initially signing up and must re-enter these same credentials when signing in, later. Our implementation follows sections 5.1.1.1 and 5.1.1.2 of the *Digital Identity Guidelines on Authentication and Lifecycle Management* published by NIST [56, Secs. 5.1.1.1-5.1.1.2], which for example specifies that password hashes must be salted and that the individual salts must be stored for each user account. A notable exception is that we did not implement a "rate-limiting mechanism" [56, Sec. 5.1.1.2], i.e. the sign-up process does not limit the number of possible failed authentication attempts.[47] The requirements and our adherence to them are discussed in Section 6.3.6 in more detail.

However, we have made some modifications respectively additions to the authentication process and its technical implementation that may require explanation. Therefore, we will only briefly summarize the processes below but detail the custom modifications. A detailed step-by-step process description can be found in Appendix J.

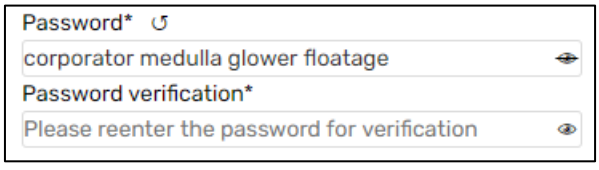

Figure 28: Password generator GUI

To create a user account, i.e. to sign up, the customer must – as on other web pages – choose an account name and specify their first and last name and optionally the email address (see Figure 17 on page 39). In contrast to other web pages, however, the customer

[47] A real-world implementation would need to utilize such a mechanism.

may not choose a password but they must generate a random password on the page itself and re-enter it to verify that it has been correctly read by them (see Figure 28). It is assumed that the customer saves the password in a secure location, e.g. a password safe.

As said, having the customer generate a password is not very common: Our personal experience is that web pages in general let the user choose a password. Hereby they usually require that the password has a certain minimum length and follows composition rules e.g. regarding the usage of letters, numbers and special characters. According to Garcia [57] one downside of this approach is that this often leads to users choosing passwords that are easy to guess by a computer but hard to remember by humans. E.g. users often modify common words by substituting letters with numbers.[57] It goes without saying that a platform like *GhostBuy*, that has a special focus on ensuring the privacy and anonymity of its customers, should seek to ensure that its customers use strong passwords: As an example, passwords that consist of the first letters of the words in a sentence[48] are easier to memorize and harder to guess than a password that is derived from a single word using a more or less foreseeable pattern. Yet, it is not possible to enforce the use of good passwords only by requiring adherence to the above composition rules and, moreover, these rules even limit the number of possible passwords. Consequently, the NIST guidelines recommend that no rules like these are enforced. [56, Sec. 5.1.1.2]

From a user perspective, Garcia proposes to use passphrases, i.e. passwords consisting of multiple words which can be easily memorized due to association but cannot be easily guessed: E.g. they should not form a semantically complete sentence and should not be the names of the user's kids.[57] The problem is that – even if the respective authentication system requires to enter several words (adding up to a certain length) – it cannot be ensured that e.g. these are *not* the names of the user's kids or pets. Also, from our personal experience, users are not – yet – used to using passphrases, therefore it might be problematic to require them to choose and enter a safe passphrase.

Therefore, our implementation generates random 4-word-passphrases from a list of 42,310 nouns. This is performed within the *Client Application*, using cryptographically secure random

[48] E.g.: Wioaghoth?Awo! = What is orange and goes hiking on the hill? – A wandarine orange!

numbers that we generate via invocation of the *Web Crypto API*.[49] We argue, that we hereby achieve a good compromise between security and usability: The passphrases are guaranteed to have at least 61 bits of entropy[50] – which could be easily increased by using longer passphrases or a larger word database – and can be entered and memorized with reasonable effort even if no password safe is used. Figure 28 shows an example of a password generated with this solution.

Please note that as of now only few browsers natively allow for unmasking password fields, which, however, is required to read and re-enter the generated password and is also recommended by NIST for entering passwords [56, Sec. 5.1.1.2]. Therefore, we implemented a JavaScript and CSS-based unmask-control that is browser-independent (see Figure 28). We also suppressed the native browser functionality (that could for example be used in Microsoft Edge), to avoid overlapping – and therefore confusing or unusable – unmask-controls.

Another derivation from likely many other authentication implementations is that we do not only hash the passwords using an individual salt (once at the client-side and once at the server side) but that we also hash the account names both on the client side and the server side. For this, however, we use one salt each for the client side and the server side, which in general are the same for all accounts, since otherwise efficient account lookups in the database would not be possible.[51] Therefore, this is to be considered not as a means of security but rather as a non-

[49] To choose the words we derive random numbers from random keys generated with the *Web Crypto API*. The API's random number generator itself is discouraged to be used for key (=password) generation, despite being a cryptographically secure random number generator (CSRNG), see https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues.

[50] $2^{61} < 42{,}310^4 < 2^{62}$ – Hereby, we assume that the Web Crypto APIs key generating function – that we use to generate the random numbers – uses a sufficiently random entropy source.

[51] The account name hashes could be changed if required, but then multiple account name hashes might have to be computed to find the account in the DB (if not all account name hashes in the DB where computed with the same salt).

critical additional effort to protect the customer's anonymity, since under normal circumstances the actual account name cannot be obtained by the *GhostBuy Platform*.[52]

Finally, it should be noted that during sign-up and sign-in the password is also used to generate the so-called *Client-side Secret Key (5-C)*[53] that is afterwards encrypted and stored in the browser-local *IndexedDB*. This AES256 key is used to encrypt the *Backend Session Key (6-B/C)* when a customer successfully completes a purchase and to decrypt it when the customer reviews previous orders. For details, please see Section 5.6.9 and Appendix J.

5.6.8. Client to Frontend Checkout Data Submission and Processing

When a customer wants to checkout the *Shopping Cart Collection*, i.e. complete the purchase process, they must specify the shipping and payment information. However, the concept of *Separation of Data* requires that the *Backend Application* does not receive this information but that it is only processed by the *Portal Application* and *Frontend Application*. Vice versa, the *Portal Application* and *Frontend Application* may not receive information about the customer's *Shopping Cart Collection* other than its price total, which needs to be known in order to process the customer's payment. (For information on how the *Shopping Cart Collection* is managed in general, see Section 5.4.)

Therefore, when the customer specifies the payment and shipping information on the *Checkout Page* (the customer must authenticate to open the page) and clicks "Continue Checkout", in summary, the following happens (a detailed description, whose steps correspond with the steps in Figure 29, can be found in Appendix K):

[52] It would only get known if someone tried to find a hash collision by computing the hashes of possible account names with the known client side-hash. It would, however, be difficult to do this since the server-side account name hash salt is secret, although in general the same for all accounts. Also, of course, finding a hash collision in general requires a lot of computational resources.

[53] AES256, derived from password using PBKDF2 algorithm with 32 byte salt and 100,000 iterations.

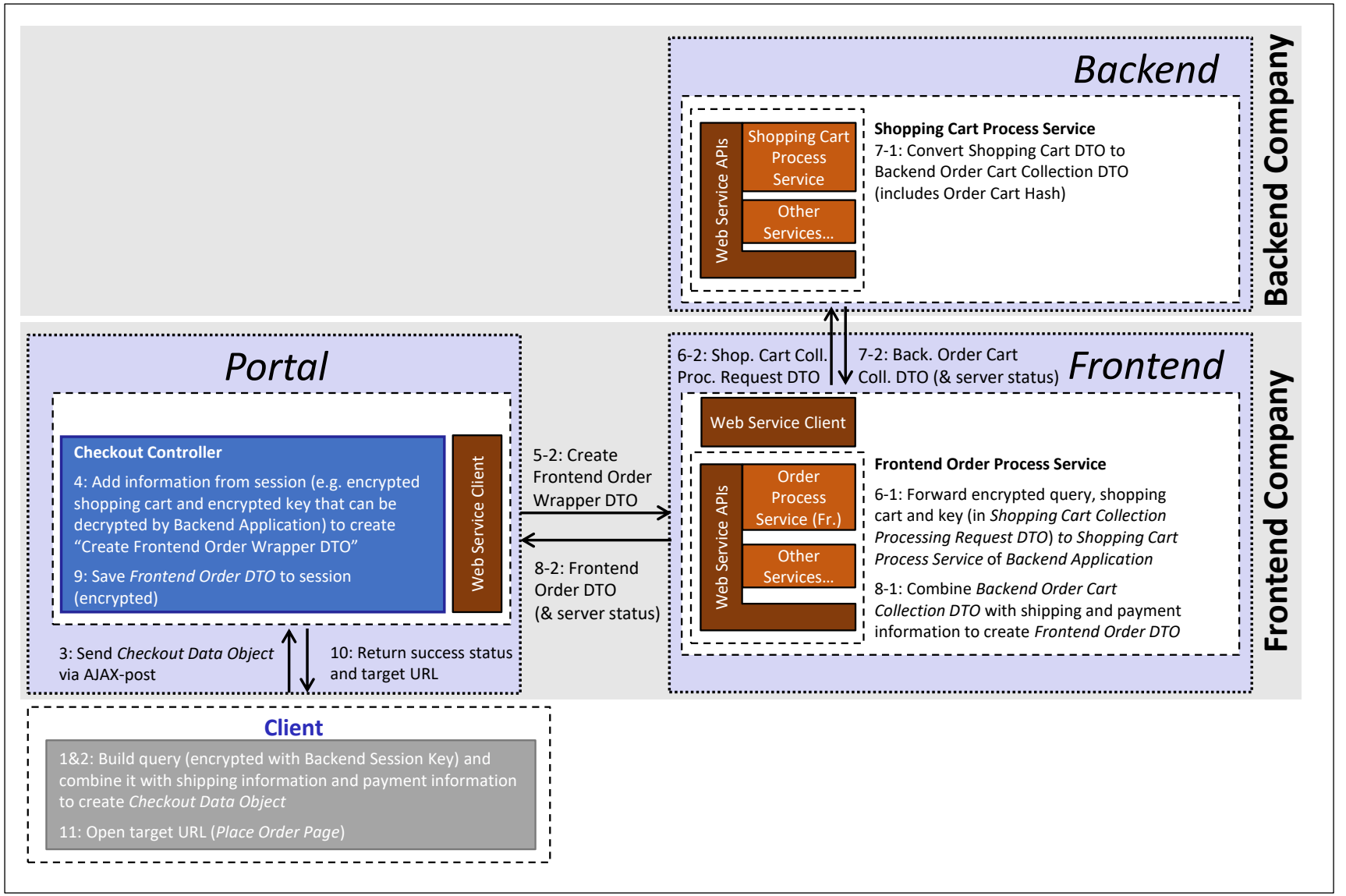

Figure 29: Client Application to Portal Application checkout data submission and processing

The *Client Application* creates a checkout data object that contains an encrypted query for the *Backend Application* and unencrypted[54] payment information (here "credit card details") and shipping information for the *Frontend Application.* This "hybrid" object is sent to the *Portal Application*, which combines it with information from the session – amongst others including the encrypted shopping cart – and forwards it to a *Frontend Application* web service for further processing. The *Frontend Application* web service forwards the encrypted query and shopping cart to a *Backend Application* web service. This web service is hereby triggered to create an encrypted *Backend Order Cart Collection DTO* that e.g. includes taxes, fees and – if applicable – converted currencies. As the only exemption to the concept of *Separation of Data* the DTO also contains the unencrypted total that has to be paid by the customer, since the *Portal Application* and *Frontend Application* have to process the payment. Please note that up to this point the *Portal Application* and *Frontend Application* never obtained unencrypted price information, so that the *Frontend Company* does not know about individual product prices in the *Shopping Cart Collection* (see Section 6.2.1. for privacy discussion).

The *Backend Application* web service also computes a salted hash[55] of the order cart information and adds it to the DTO to allow for later authorisation of the order by the *Client Application* (see next section). After the *Frontend Application* web service received the *Backend Order Cart Collection DTO* from the *Backend Application* web service it combines it with the payment and shipping information to create a so-called *Frontend Order DTO* that it saves to the session. The *Client Application* will later obtain this object to – where necessary – decrypt and display all order-related information. During the above process the *Portal Application* and *Frontend Application* never obtain shopping/order cart-related information (other than the price total) since this information is encrypted with the *Backend Session Key (6-B/C)*.

---

[54] Although the payment and shipping information is unencrypted it is secure since it will be transmitted over secure connections and it is encrypted when saved to the *Frontend Application's* database.

[55] In our implementation we use the PBKDF2 algorithm for this (with 1 iteration) since that saved us implementing a separate class for the hashing function in the *Backend Application*. In the *Client Application* we already had implemented a separate hash function for generating the URL-ID, however, we also used the PBKDF2 algorithm for this, since that was used for other purposes as well.

Please note: In our prototype implementation we also include the credit card currency in the encrypted query parameters that are sent to the *Backend Application* so that it can correctly convert the shopping carts' totals (for each shop in the *Shopping/Order Cart Collection*). However, this should be avoided since the customer's credit card currency is sensitive information and should actually not be communicated to the *Backend Application*. We noticed this problem while describing this process. However, with regard to the submission deadline, changing the implementation was no longer possible at this point. Therefore, we briefly want to describe how this problem could be solved to demonstrate that this is not an unfixable design flaw:

- To fix the problem, the customer`s credit card currency is not included in the encrypted query so that the *Backend Application* does not obtain this information. Therefore, it converts all the individual shop's totals, the fees, taxes and the overall total to a pre-defined currency, e.g. CAD, when building the *Order Cart Collection*. It includes the respective conversion rate into the encrypted *Order Cart Collection DTO* that it returns to the *Portal Application*. The only unencrypted information is the *Order Cart Collection's* overall total (which is also in CAD).

- When receiving the *Order Cart Collection DTO*, the *Portal Application* converts the overall total (CAD) into the customer's credit card's currency so that it can be correctly displayed to the customer in the *Client Application*. Hereby the respective CAD-to-credit-card-currency-conversion-rate is also transmitted to the *Client Application.*

- The *Client Application* decrypts the *Order Cart Collection DTO* that it obtains from the *Portal Application*. To recall, apart from the product information this DTO includes the individual shops' totals, the respective to-CAD-conversion-rates, the CAD-to-credit-card-currency-conversion rate as well as the fees and taxes. Thus, the *Client Application* can calculate and display the individual shops' totals as well as the fees, taxes and overall total in the customer's credit card currency. Hereby, it would only display the original price and the converted price but not the intermediate CAD price so that the order layout would be equivalent to the prototype implementation's layout. Please note that if the shop currency and/or the customer's credit card currency are CAD, the respective conversion rate to the intermediate CAD price is 1:1.

5.6.9. Order Data Merging, Authorization, History Preparation and Placement

When the *Client Application* opens the *Place Order Page* it triggers a process that continues the process described in the previous section. The process can be summarized as follows (for details, please see the first process description in Appendix L):

Using AJAX requests, the *Client Application* obtains the *Frontend Order DTO*, i.e. the shipping and payment information as well as the encrypted *Order Cart Collection* information, from the *Portal Application*, which had previously stored this information in the web session. Subsequently, the *Client Application* decrypts the *Order Cart Collection* with the *Backend Session Key (6-B/C)* and populates the *Place Order Page* with all obtained information[56]. It also hashes the *Order Cart Collection*, using the same salt that was used by the *Backend Application* when it computed its hash of the *Order Cart Collection*. This *Client Application* hash of the order cart is used as an authorization token for finally placing the order as described below.

When the customer clicks the "Place your order" button on the *Place Order Page*, in short, the following process is triggered (illustrated in Figure 30):[57]

The *Client Application* retrieves the encrypted *Backend Session Key (6-B/C)* from the *IndexedDB*, decrypts it and then re-encrypts it with the *Client-side Secret Key (5-C)* that was derived from the user password during authentication and which was stored in the *IndexedDB*, too. Then, the *Client Application* sends this encrypted *Backend Session Key (6-B/C)* and the client-side hash of the order cart to the *Portal Application*. The *Portal Application* checks the authorization of the order by comparing the *Client Application's* order cart hash to the *Backend Application's* order cart hash. If the hashes match, the order was successfully authorized by the customer[58] and the order is (tried to be) placed, subsequently.

---

[56] *Order Cart Collection* information as well as the shipping and payment information.

[57] For details, including how unsuccessful order authorizations are handled, please see the second process description in Appendix L whose steps correspond with the steps in Figure 30.

[58] See Appendix L for description of unsuccessful authorization handling.

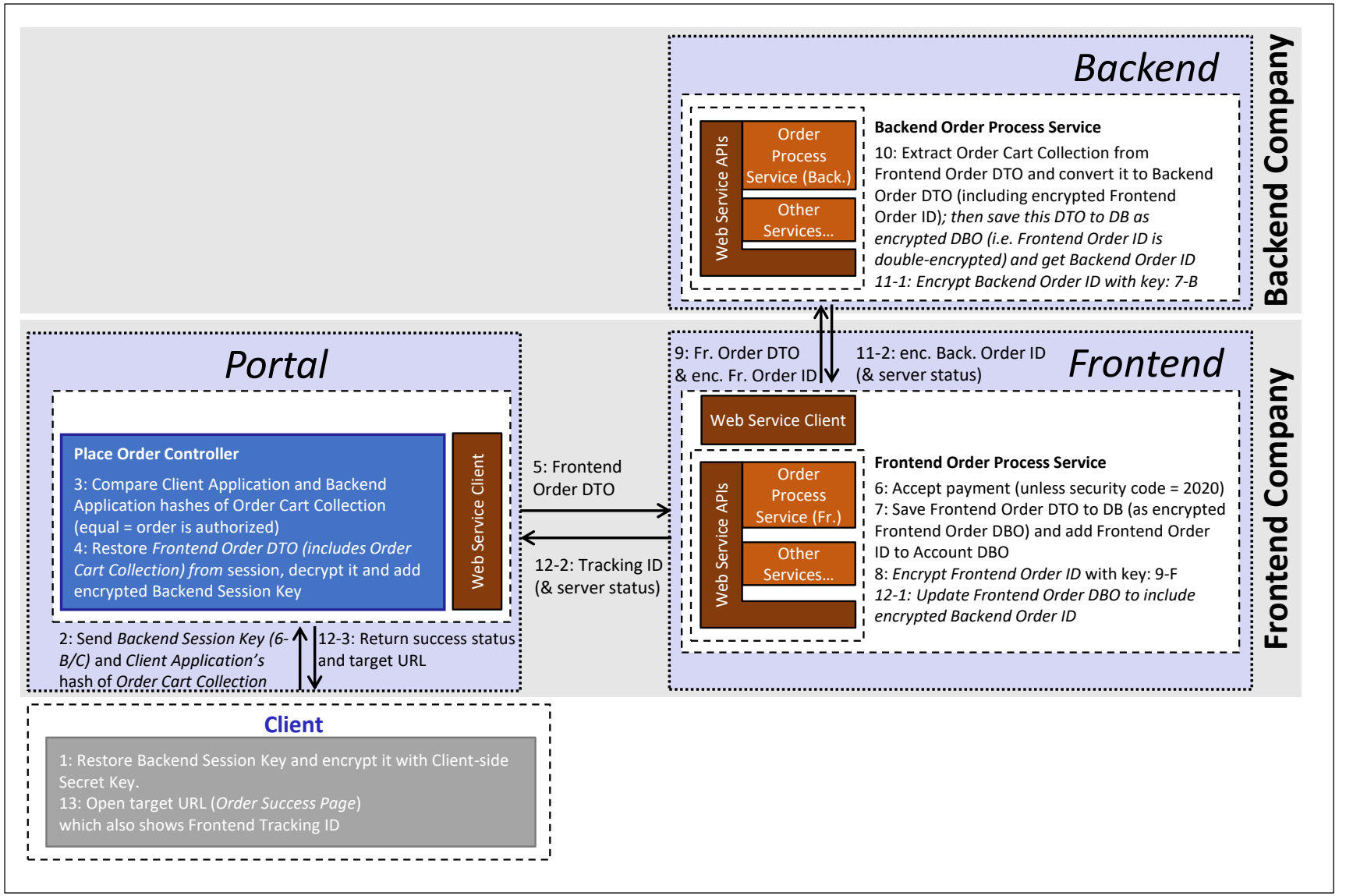

Figure 30: Order data merging, authorization, history preparation and placement

The *Portal Application* combines the encrypted *Backend Session Key (6-B/C)* with the frontend order information that it restores from the web session. Due to this, the customer will later be able to decrypt the order information – when reviewing it in the *Order History Page* – since the *Client-side Secret Key (5-C)* can be regenerated during any subsequent sign-in and can therefore be used to decrypt the *Backend Session Key (6-B/C)*. The *Portal Application* subsequently triggers a web service of the *Frontend Application* to check the payment information and place the order. The *Frontend Application's* web service will accept the payment information unless – for demonstration purposes – the credit card security number is "2020".[59] Please note that in a real-world implementation the payment e.g. would be processed by invoking a third-party web service.

When the payment is accepted, the *Frontend Application* web service saves the frontend order information to the *Frontend Application* database (as encrypted *Database Object (DBO)*, see Section 5.8). The hereby assigned *Frontend Order ID*[60] is added to the respective encrypted *Account DBO,* so that later all orders belonging to that account can be retrieved.

The *Frontend Application* web service also triggers a web service of the *Backend Application* to actually order the respective products at the e-commerce website (e.g. eBay). Please note that in the prototype the ordering of the products is not actually performed but only written to a log file. The *Backend Application's* web service manages the related information in a so-called *Backend Order DTO and DBO.*

In an early stage of the thesis, we considered using one way-functions to link the *Frontend Order* to the *Backend Order.* During the design and development phase, however, we decided against this, since e.g. one would have to use secret components for these functions, which – if ever changed – would require updating all these references. Overall, the one-way-function-approach seemed much more complex and error-prone than the following, implemented approach which is very similar to the "Virtual Session" approach described in Section 5.4:

[59] See the second process description in Appendix L for details regarding a failed payment authorization.

[60] Assigned by the invoked *Data Access Object (DAO) Implementation*

The *Frontend Order ID* is used as reference ID to link the *Frontend Order* to the associated *Backend Order*. Vice versa, the so-called *Backend Order ID* is used as reference ID to link the *Backend Order* to the original *Frontend Order.* To protect the IDs – and thus the link – from unauthorized access, each ID is encrypted by the originating application[61] and then stored inside/alongside[62] the respective other order (*Remote Order ID*). For a detailed description of how this is done please see steps 8 and following of the second process description in Appendix L.

When the *Backend Order* has been triggered by the *Frontend Application's* web service and the link between the two orders has been created, i.e. when the order has been placed, the *Portal Application* returns the target URL[63] (*Order Success Page*) to the *Client Application.* Accordingly, the *Client Application* opens the URL of the *Order Success Page*. Meanwhile, the customer is signed out by invalidating the session. This ensures that the *Backend Application* cannot – without further ado – attribute subsequent searches, purchases etc. to the same – unknown[64] – customer (see Section 6.2.2).

Please note that if the customer later wants to see their previous orders, this is – in general – handled by the *Portal Application* and *Frontend Application* alone: The encrypted *Order Cart Collections* are contained in the *Frontend Order DBOs* that are associated with the respective account. They can be decrypted in the Client Application (*Order History Page*) when the "owning customer" signs in. To recall, this is possible due to the following: The *Order Cart Collections* were encrypted with the *Backend Session Key (6-B/C)* which is unknown to the *Portal Application* and *Frontend Application*. This key, however, was saved as part of the *Frontend Order DTO/DBO*, though encrypted with the *Client-side Secret Key (5-C)* which can be derived from the customer's password. I.e. the *Frontend Company* (*Portal Application & Frontend Application*) only knows how many orders a customer has completed and what

---

[61] Performed by web services of the *Frontend Application* and *Backend Application* with the *Frontend Application Web Service Key (*9-F) and *Backend Application Web Service Key (*7-B*), respectively.*

[62] The respective *DAO Implementations* save the order DTOs and IDs as encrypted Java objects that we call DataBase Objects (DBOs).

[63] The URL, that the *Client Application* has to open now (since the order was successfully placed).

[64] Unknown to the *Backend Application*.

overall price they had to pay for each order but it does not know the content of the orders such as the particular products purchased. Nevertheless, once a purchase has been completed, the customer can review its details in the *Order History Page* without involvement of the *Backend Application*. To demonstrate the feasibility of this approach, as part of the *GhostBuy Prototype*

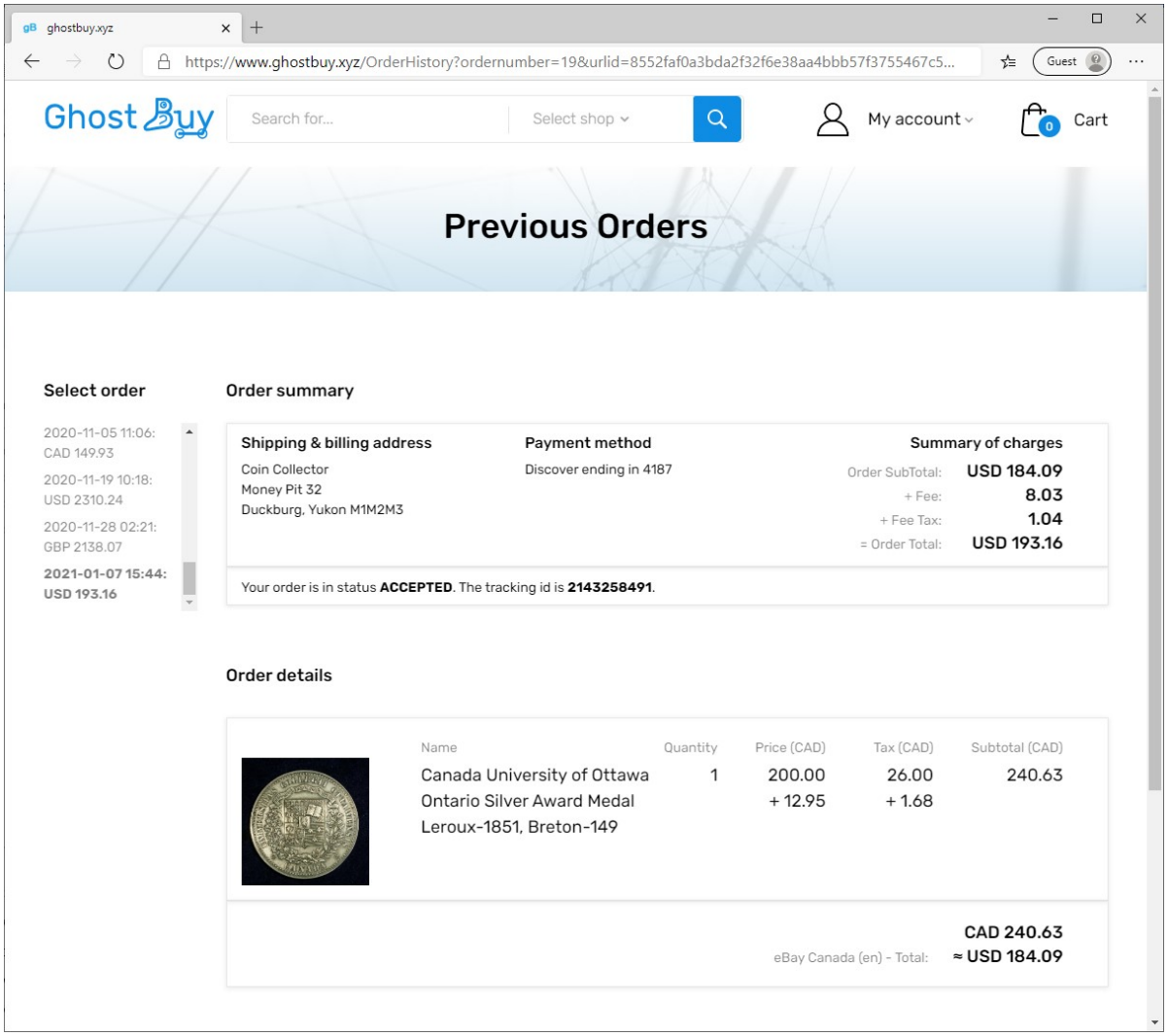

Figure 31: Order History Page

we implemented the *Order History Page* (see Figure 31) as well as the necessary controllers, web services etc.

Yet, if a customer would forget their password it would still be possible to reset it: In this case, however, the *Backend Application* would have to re-encrypt all *Order Cart Collections* for the respective customer, e.g. with the *Backend Session Key (6-B/C)* that is used during the very session in which the customer resets the password. For re-encrypting the *Order Cart Collections*, the *Frontend Application* would have to pass them, one-by-one, to the *Backend*

Application since this application does not know which orders belong to the same customer. The current/new *Backend Session Key (6-B/C)* would be encrypted with the *Client-side Secret Key (5-C)* and saved with the *Frontend Order DBO/DTO* as usual. Yet, due to time restrictions and since this feature is not crucial to demonstrate a working prototype, we did not implement the password reset process in the prototype.

5.7. Backend to Frontend Workflow Outline and Prototype Support

While the previous sections 5.6.1 through 5.6.9 covered the data processing and data flows that take place to provide the directly customer-related functionality, this section will briefly outline how ordered packages would be shipped and handled in order to preserve the customers' anonymity and privacy. Figure 32 illustrates this process:

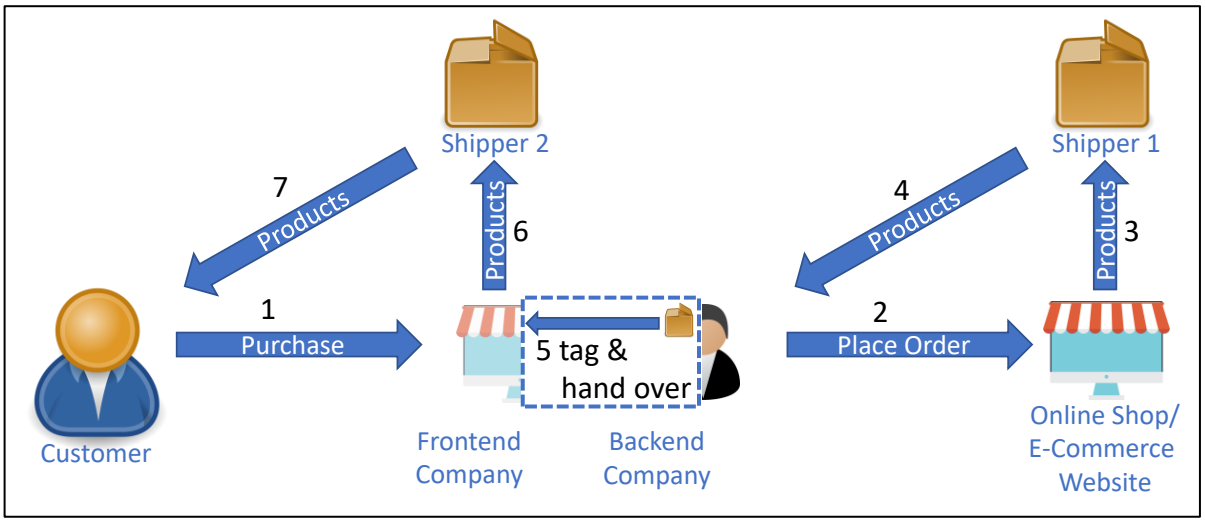

Figure 32: Purchase, order and Backend Company to Frontend Company package handover & shipping

When the customer successfully completes a purchase on *GhostBuy*, the *Backend Application* triggers the related order(s) on the respective e-commerce website(s) – which in our prototype implementation is only simulated by being written to a log file. Hereby the *Backend Application* specifies a so-called *Backend Tracking ID* as part of the shipping address – *e.g. as "c/o"* – which can be used to identify the package, respectively link the package to the associated order. We implemented the *Backend Tracking ID* as a random, unique 4-byte integer but for example it also could be implemented as a random, unique name in case a certain e-commerce website would not allow for specifying the *Backend Tracking ID* in the shipping address otherwise. The *Backend Tracking ID* can be reassigned to a new order when the related package has been successfully received and forwarded to the customer. This stands

in contrast to the *Backend Order ID*, which is an 8-byte integer ("long") and will never be reassigned.

The e-commerce website, i.e. the online shop, will ship the package to the *Backend Company*. Once the package is received, an employee of the *Backend Company* will disguise the package's origin and original shipping information, for example by wrapping the package into (self-adhesive[65]) paper or by opening the package and placing the products in a new package – if possible without being able to see the products.[66] Hereby the employee will record the *Backend Tracking ID* beforehand, respectively enter it into an input mask that is only available to employees of the *Backend Company*. This triggers a lookup of the associated *Backend*

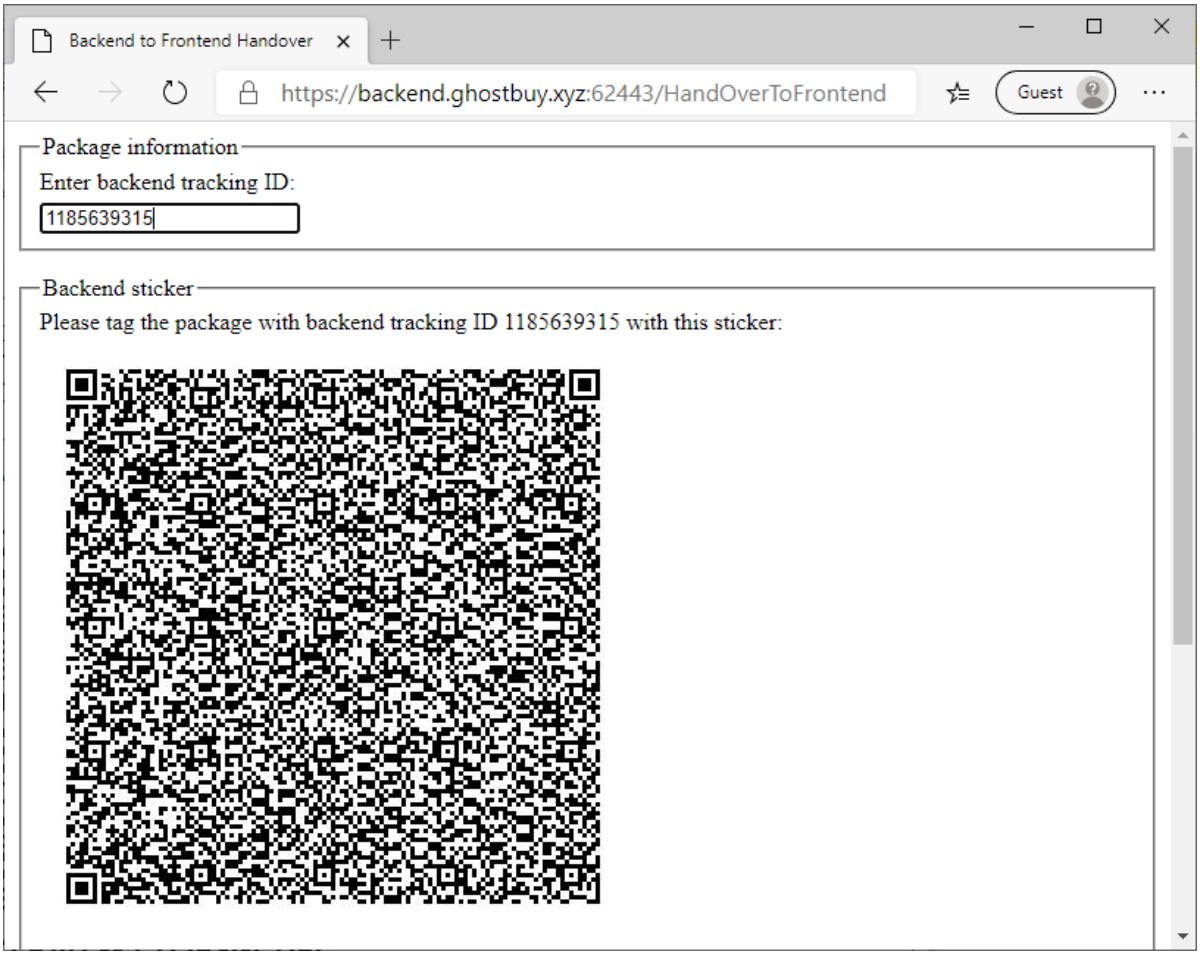

Figure 33: Backend Company to Frontend Company handover - encrypted Frontend Order ID as QR code

[65] To avoid that someone accidentally can see the original packaging when opening the package.

[66] If employees of the *Backend Company* do not know the products, successful correlation attacks would not harm the customer`s privacy as much as if the products were known (see Section 6.2.1).

Order DBO in the *Backend Application's* database, and returns the encrypted *Frontend Order ID* recorded therein. Hereby the Backend Application's database "outer" encryption is decrypted, but the *Frontend Order ID* is still encrypted with the *Frontend Application Web Service Key (9-F)*.

The employee places the encrypted *Frontend Order ID* on the package in a non-human-readable format. We implemented the *Backend Company*'s input mask as a rudimentary, though working, web page which outputs a QR code (see Figure 33). To make testing easier, our web page additionally displays the textual representation of the encrypted *Frontend Order ID* that is encoded in the QR code (see Figure 34).

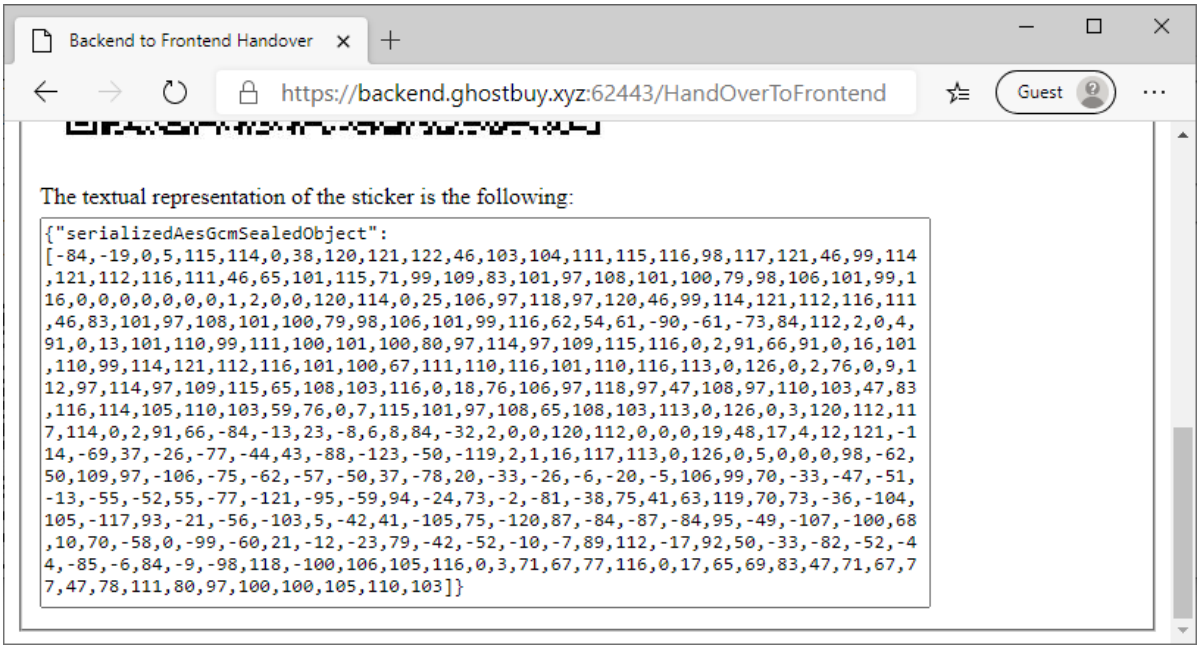

Figure 34: Backend Company to Frontend Company handover - encrypted Frontend Order ID as text

Please note, although the *Frontend Order ID* is encrypted and QR codes should be hard to memorize, a real-world implementation might for example use RFID chips to make it even harder for employees to record/memorize it, in order to mitigate the risk of correlation attacks (see Section 6.3.5).

After the encrypted *Frontend Order ID* has been placed on the package, the package is handed over to the *Frontend Company.* Here, an employee scans the QR code and enters its content, i.e. the textual representation of the encrypted Frontend Order ID, into another input mask – which of course only is available to employees of the *Frontend Company*.

We also implemented the *Frontend Company*'s input mask as a rudimentary, though working, web page. When the encrypted *Frontend Order ID* is entered into the input mask, it is parsed, decrypted and the associated *Frontend Order DBO* – respectively the shipping information contained therein – is retrieved from the *Frontend Application's* database and shown on the web page (see Figure 35). Then, the *Frontend Company's* employee ships the package to the displayed address.

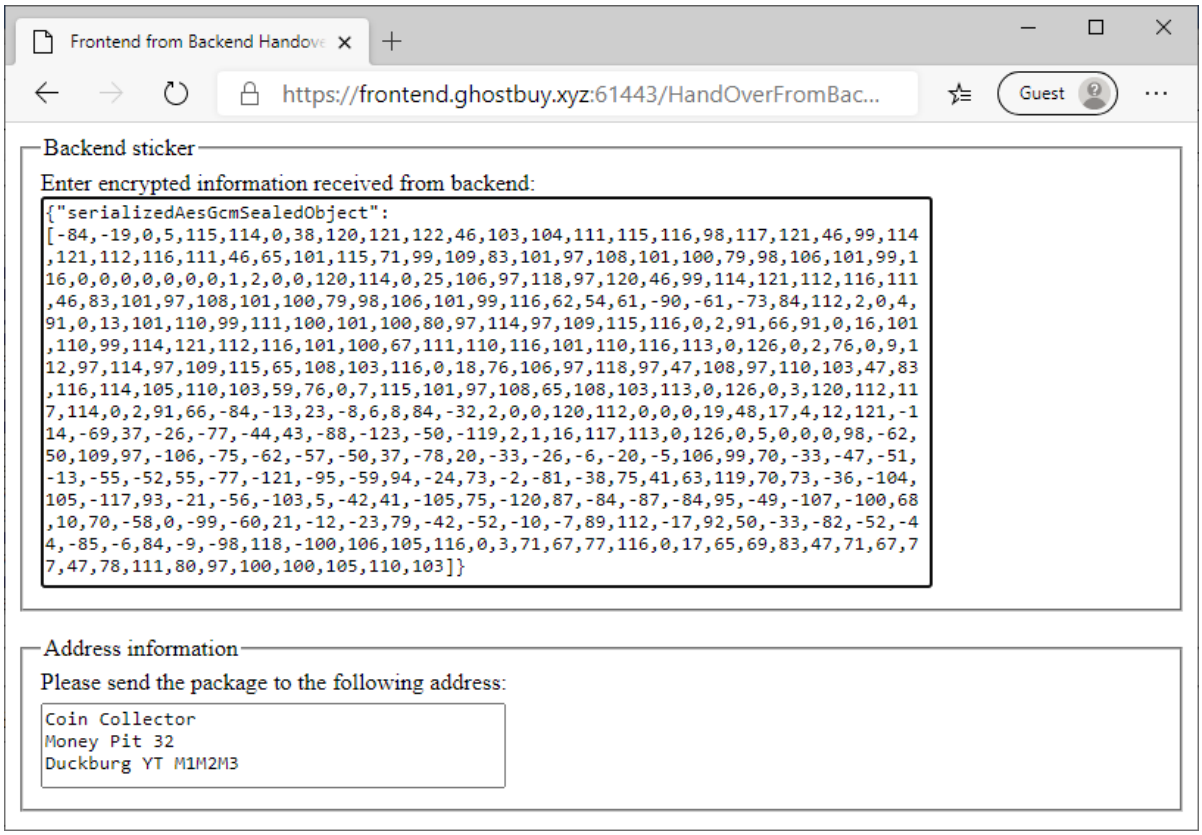

Figure 35: Frontend Company from Backend Company handover - shipping information retrieval

Please note that to realize the input mask web pages, we had to implement controllers at the *Frontend Application* and *Backend Application*. This differs from the controllers for the directly customer-related functionality which are all implemented in the *Portal Application* alone. In our prototype implementation we placed the *Frontend Application* and *Backend Application* controllers on the same web servers as the respective application's web services. A real-world implementation might use dedicated servers for this purpose.

5.8. Database Implementation

As can be seen in the previous sections, our *GhostBuy Prototype* implementation encrypts sensitive data whenever it is stored in the session / virtual session by the *Portal Application* or in the browser-local *IndexedDB* by the *Client Application.* We also described that the *Frontend Application* and *Backend Application* persistently store data in dedicated MySQL databases. Hereby we use *Hibernate* to implement the *object-relational mapping (ORM)* that allows for storing Java objects in the databases. Consequently, we also sought to encrypt the information that is saved in the databases. To achieve this, we considered the following three options:

1. The least complex but also least secure option to encrypt the database information is to encrypt the hard disks on which the database files are stored. In fact the hard drives on which database files are located are Bitlocker-encrypted as a general security measure, but this alone does not ensure a high level of security regarding the information that is stored in the databases: For example, when the operating system of the development computer is running and thus the Bitlocker-encrypted hard disks are unlocked, any sufficiently privileged user (local or remote) can read or copy the database files. If the database files contain plaintext information, this information would not be secure. Accordingly, another layer of encryption is required:

2. The second option to encrypt the database information is to encrypt it on the database level itself. Hereby one possibility is the so-called data at rest encryption that transparently encrypts the entire database when written to or read from the hard disk.[67] Another possibility is to encrypt individual columns of the database by utilizing database specific encryption methods.[68] However, encryption at the database level has the general disadvantage that it has to be configured/implemented for each database engine individually which means that changing the database engine later would be problematic. Also, this type of encryption would be hard to use in combination with Hibernate (or any other ORM solution) especially when Hibernate defines the database schema - which it mostly does in our prototype implementation. Another security

[67] Transparent Data Encryption (TDE): https://www.mysql.com/products/enterprise/tde.html

[68] MySQL Encryption Functions: https://dev.mysql.com/doc/refman/8.0/en/encryption-functions.html

related problem is that if the encryption is performed at the database level, people with sufficient access privileges to the database (database administrators) can access and decrypt the information stored in the database.

3. Therefore, the third option that we considered for encrypting database information is to perform the encryption at the application level. This option does not have the downsides of the previous two. Yet, the fact that the encryption has to be realized in the application itself would be a disadvantage in the context of other applications that do not have a focus on encryption. In the context of our prototype implementation, on the other hand, it was comparatively straightforward to realise, as we already used object encryption within the application. More details of the actual implementation are explained below:

The Hibernate-based object relational mapping that we use to write Java objects to the MySQL databases of the *Frontend Application* and the *Backend Application* in general follows the same pattern that was used for implementing the previous student bookshop project.

Each object that must be saved to and read from the database is realized as a triplet of three Java classes:

1. The to be saved object is defined in a so-called *DataBase Object (DBO)* which otherwise is often referred to as a "bean". This class defines the attributes of the object which themselves can also be objects (e.g. DTOs), and for example also if these attributes may be null. It also specifies certain database related properties such as the primary key, the data type an attribute shall be mapped to (if not properly auto-detected by Hibernate) and how the association with other DBOs shall look like (e.g. 1:1, 1:n).

2. The interface (abstract class) *Data Access Object (DAO)* specifies which methods we need to write and read the DBO to and from the database. For example, the DAO can specify that a certain method shall find and return the respective DBO based on the ID which is given as input parameter and that another method shall create and save a DBO with a DTO be the input parameter.

3. Finally, the *Data Access Object Implementation (DAO Implementation)* implements the methods that are defined in the DAO interface. *The DAO Implementations* hereby

also perform the encryption and decryption of the sensitive information. Since all *DAO Implementations* of one application are based on the same parent class they share a common encryption key which is imported by the respective parent class (*Backend Application DAO Implementation Key (8-B)* and *Frontend Application DAO Implementation Key (10-F))*

Describing all the details of the DBOs, DAOs and *DAO Implementations* of the prototype implementation is not feasible (nor necessary) within this thesis. However, Figure 36 shows the Frontend Application's database schema and hereby the most important aspects of the database implementation (for more details regarding the implementation please see the provided source code and documentation):

- All sensitive, encrypted information, that we store in the database is realized as a so-called *AES-GCM-Sealed-Object DBO* which contains a custom encrypted (sealed)

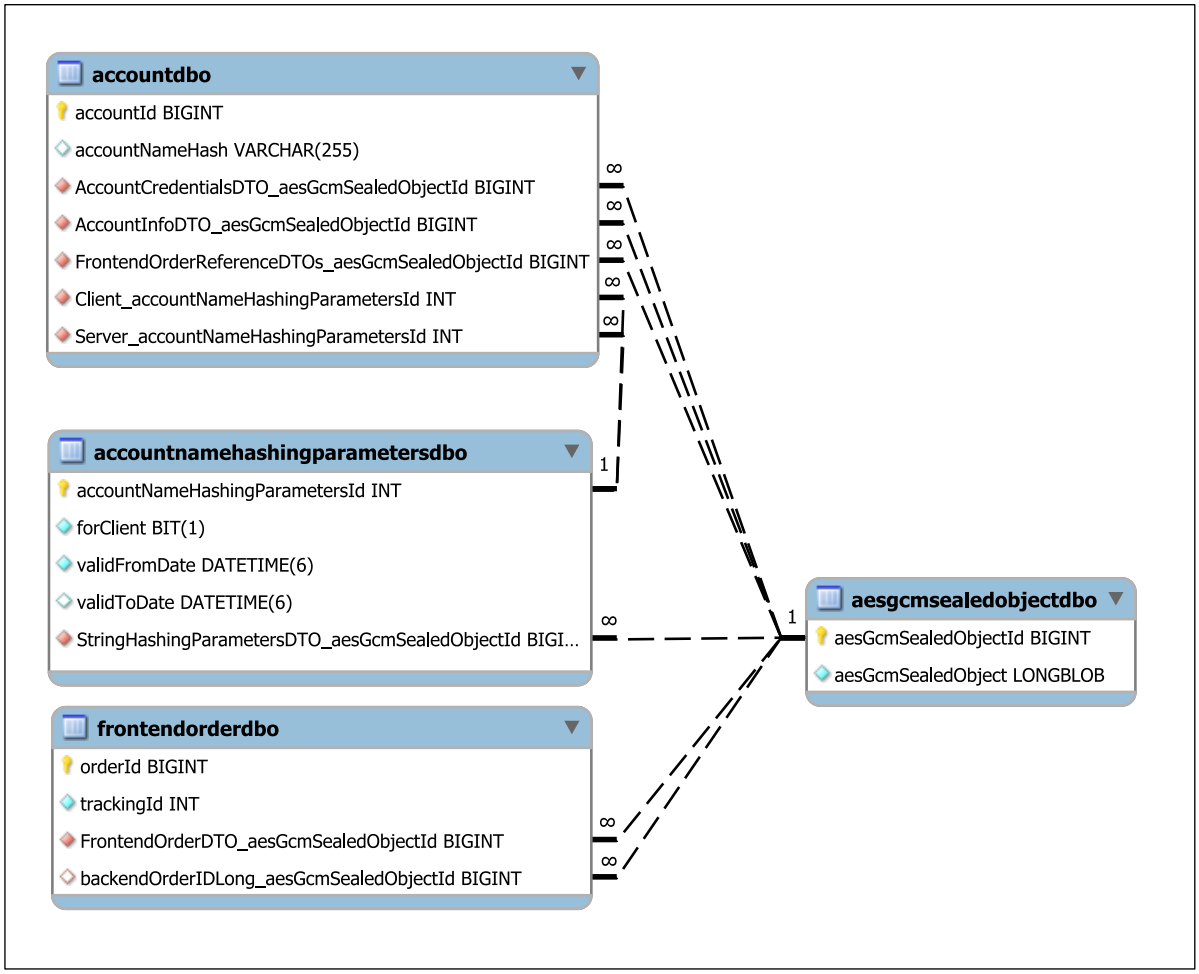

Figure 36: Frontend Application database schema

Java object. The table representation of this DBO uses a BLOB to store the encrypted information.

- All other DBOs, have one or more *AES-GCM-Sealed-Object DBO* as member objects in which the sensitive information is encrypted and stored, usually in the form of custom DTOs.

- Within the database table for the *Frontend Order DBO* one can see that amongst others it also contains a *Backend Order ID* (see Figure 36). This ID is the reference ID to link the *Frontend Order DBO* to the *Backend Order DBO* which is stored in the *Backend Application's* database (see Section 5.6.9). The *Backend Order ID* is encrypted twice, first with the *Backend Application Web Service Key (7-B)* and afterwards with the *Frontend Application DAO Implementation Key (10-F).*

- Although not visible in the diagram we want to point out that most object IDs – except for the IDs of the *Accountname Hashing Parameters DBO* – are generated randomly in order to prevent any conclusions to be drawn for example about the respective object`s time of creation. To have full control over the process, including the random number generator that is used for this, the randomization has been implemented in the respective *DAO Implementations*. If required, the same could be realized for the IDs of the *Accountname Hashing Parameters DBO*, however, this would be less efficient if account name hashing parameters were to be changed at a certain point of time (as there would be no defined order for iterating through the previous hashing parameters).

The *Backend Application's* database schema is similar to the *Frontend Application's* database schema. However, since account information is not handled or stored by the *Backend Application* this schema only contains two tables: The first table is for the *Backend Order DBO* which is similar to the *Frontend Order DBO* and the second table is for the equivalent *AES-GCM-Sealed-Object DBO*. Figure 39 in Appendix N shows the *Backend Application's* database schema.

5.9. Important Prototype Properties and Features

Regarding the *GhostBuy Prototype* implementation, we sought to create a complete and convincing solution which should consider and realize not only the basic functionality but also

pay attention to other important details of such an application and service. To be clear, as described in the previous sections, we do not claim that the prototype architecture and implementation is perfect in any way (see also Section 6.5). Yet, by considering the complete platform ecosystem and also paying attention to certain GUI features we wanted to convince the reader that an *All-Steps Anonymous Purchase Platform* similar to *GhostBuy* can indeed be realized with an appealing interface and usability.

Therefore, below we would like to summarize important prototype properties and features that we described so far and subsequently will briefly describe additional aspects of the *GhostBuy Prototype* which we think are important to know.

The following properties and features were described previously:

- Separation of Data, providing anonymity/privacy (see also Section 6.2)
    - Ability to load encrypted content without revealing it to the *Frontend Company* (*Portal Application* and *Frontend Application*), including product images, thereby maintaining a responsive user/customer experience
    - Ability to (rudimentary) sanitize “original product description”, i.e. to remove external resources
    - Ability to specify search terms and apply filters for product searches without revealing these to the *Frontend Company*
    - Ability to check search terms for potentially contained personal information
    - Ability to efficiently preserve/restore the client state, based on the browser URL, but hereby not leaking information to the *Frontend Company* and not allowing to (e.g. accidentally) share any information (with regards to the URL) with anyone
    - Ability to review previous orders without involving the *Backend Company* which therefore cannot link orders to the same customer (see also Section 6.2.2)
- Secure authentication, including the generation of secure passwords.

- Correct detection of the authenticated state on *Sign-Up Page*, *Sign-In Page* and *Sign-Out* Page even when navigating backwards (see Section 5.3)
- Order authorization, based on (salted) hashes to ensure and prove a correct order information transfer (and to avoid accidentally ordering a wrong shopping cart as described in step 3 of the second process description in Appendix L)
- Permanent storage of account and order information in dedicated databases via application-level encryption and storage of Java objects (Hibernate-based)
- Outline of the package handover process from the *Backend Company* to the *Frontend Company* and rudimentary, but functional implementation of the input masks required for this (opposite direction could be realized similarly)

In addition to the above we would like to briefly describe the following aspects of the prototype:

- The *GhostBuy Prototype* has a responsive user interface which is – in addition to loading content dynamically after page load – also achieved by the following:
  Objects/information such as the shopping cart item count (shown in page header) and the shopping cart content are saved, encrypted, in the virtual session and are not updated or re-encrypted as long as they do not change. Therefore, this information can be retrieved from the *Portal Application* and displayed very quickly.
- The Client Application has to retrieve the *Client Wrapping Key (2-P/C)* from the *Portal Application* to encrypt and decrypt information that is stored in the browser-local *IndexedDB*. Since this often happens several times in quick succession for one page load, we implemented a "Memoization" function that implicitly caches the key, with concurrent requests being transparently resolved by the function. We adapted the function from a different use case[69] and improved it so that the key is automatically deleted from memory whenever it has not been requested for more than 1 second.

[69] https://medium.com/@bluepnume/async-javascript-is-much-more-fun-when-you-spend-less-time-thinking-about-control-flow-8580ce9f73fc

- On the Shop Page, the default sorting option applies to all products, as the sorting-preference is transmitted to the *Backend Application* (via the *Portal Application*) which requests accordingly sorted product lists from the e-commerce platform (eBay). In addition, however, there is also a "sort in page"-option which only sorts the products that are displayed on the *Shop Page* in that very moment (e.g. products 21 to 40). Hereby animations illustrate the sorting process (see Figure 40 in Appendix O). The in-page sorting functionality has been adopted from the OneTech template that we used to implement the *GhostBuy Prototype*. We did so to demonstrate that once the encrypted content has been decrypted and populated on the page, it is technically the same as regular, unencrypted content and can be manipulated and animated using common web technologies such as HTML5 and JavaScript.

- The *GhostBuy Prototype* can be displayed and used fairly well with mobile device browsers as for example the mobile Chrome browser. This is partly due to the OneTech template that includes "responsive" CSS files for mobile devices. In addition, however, this also shows that even average mobile devices have enough computational power and that at least certain mobile browsers have sufficient capabilities (support of *Web Crypto API*) to use a platform like *GhostBuy*.[70] Yet, since using the *GhostBuy Prototype* with a mobile device browser was not a main goal of the implementation, we did not pay much attention to that aspect, causing the layout of the web page to be sometimes not quite coherent on mobile devices (see next bullet point).

- The *GhostBuy Prototype* has been successfully tested with the following browsers:
  Desktop:
  - Microsoft Edge (Chromium-based): Version 87.0.664.75 (Official build) (64-bit)
  - Mozilla Firefox EU: Version 84.0.2 (64-Bit)

[70] We were able to use all important functionality of the web page on these mobile browser and smartphones: Chrome on Google Pixel 2 (Android), Safari on iPhone 6S (iOS) and Samsung Internet on Samsung Galaxy S10e (Android).

  - Google Chrome (Chromium-based): Version 87.0.4280.141 (Official Build) (64-Bit)
  - Opera (Chromium-based): Version 73.0.3856.344

  Mobile (runs with layout issues):
  - Google Chrome (on Android): Version 87.0.4280.141
  - Samsung Internet (on Android): Version 13.0.2.9
  - Apple Safari (on iPhone/iOS): Version 14

- Finally, the *GhostBuy Prototype* provides protection against certain mis-authentication attacks as described in Section 6.3.6.

5.10. Package Verification and Law Enforcement Inquiries

We optionally planned to implement the following three features in the *GhostBuy Prototype* (depending on the COVID-19 related situation and on how fast the implementation would progress):

a. Hash-based verification of transactions
b. QR code or alternative verification of received packages (package seal)
c. Query interface(s) for law enforcement authorities

As described in Section 5.6.9 we were able to implement (a), but there was not enough time for (b) and (c). However, we pledged to outline an implementation of above features in this case.

Regarding the verification of received packages (b) it must be understood that the purpose of this is twofold: First, the customer shall be reassured that they received the correct package and that it had not been opened by *Shipper 2* during the delivery from the *Frontend Company* to the customer. Second, the *Frontend Company* shall get a confirmation of this, too, for example to verify the reliability of *Shipper 2* and to identify if customers are (likely) dishonest. We envisioned the process as follows: Before the *Frontend Company* ships the package to the customer, it will remove the *Backend Company's* QR code (or RFID chip, see Section 5.7) and physically seal the package with, for example, a tape (and a seal) that cannot be tampered with without someone noticing (see Figure 42 and Figure 43 in Appendix T). Additionally, it

will print the package`s *Frontend Tracking ID* onto the seal, e.g. in form of a QR code (for the customer's convenience).[71] When the customer receives the package, they would have to open an app or a special *Portal Application / Client Application* web page – requiring them to authenticate with account name and password – and enter (or scan) the *Frontend Tracking ID*. The app / web page would then verify that this is the correct package – e.g. by retrieving and displaying the product information – and the customer would verify the proper receipt of the (unopened) package. The verification process should incorporate cryptographic functions (electronic signature) that verify the authenticity of the messages (package content & verification), the customer and *Frontend Company*.

The purpose of the query interface for law enforcement authorities (c) is to enable them to investigate purchases made by customers of the *GhostBuy Platform*, which is required by law, e.g. to fight money laundering or other illegal activities. However, since the main goal of the *GhostBuy Platform* is to protect the privacy and anonymity of its customers, this interface must ensure the following: First, the law enforcement authorities may only obtain information that they are authorized to obtain, which means they may not obtain unrelated information about other customers and purchases. Second, the *GhostBuy Platform* must not learn which information was (or was not) retrieved or even only queried by the law enforcement authorities.

This can be achieved with the following process:

1. The *Frontend Company* and law enforcement authority use *Private Set Intersection (PSI)* – which is for example described in [58] – to detect if the *Frontend Company* has any information of interest to the authority. As a prerequisite, it must be ensured that the dataset of the law enforcement authority is minimal (i.e. does not contain any unrelated records), which can be certified, for example, by a digital signature of the judge who ordered the search warrant. Also, a PSI-protocol should be chosen that will only allow the law enforcement agency to learn the matching records but will not allow

[71] Obviously, it must be ensured that *Shipper 2* cannot simply repackage the products in another packaging, which would require him to imitate the original packaging and seal. This could be achieved, for example, by using a GhostBuy-branded packaging and/or tape and seal.

the *Frontend Company* to learn it. If there is no match, the process stops, otherwise it continues with step 2.

2. Since in step 1 the law enforcement authority for example only had a name of a customer or the *Frontend Tracking ID* of an order, in step 2 the authority will obtain all related information from the *Frontend Company.* For this the *Frontend Company* and the law enforcement authority use an efficient *Oblivious Transfer (OT) Extensions Protocol*. This will allow the law enforcement authority to learn all the details about the respective purchase records without it learning anything about other purchases or the *Frontend Company* learning which records were or were not retrieved.

3. In an optional third (and fourth) step, the law enforcement authority will repeat step 1 and 2, but this time interacting with the *Backend Company*, in order to retrieve the product / online shop order related information for the respective purchases. However, as of now, an intermediate step would be necessary: To recall, an encrypted *Backend Order ID* was previously provided by the *Backend Company / Application* and stored in the *Frontend Application* database (see Section 5.6.9), but since it has been encrypted with a random IV (and this IV and encrypted *Backend Order ID* was not stored in the *Backend Application* database) it cannot be used to perform a *Private Set Intersection.* The same applies for the *Frontend Order ID* that was encrypted by the *Frontend Company / Application* and stored in the *Backend Application* database (see Section 5.6.9). Therefore, as the intermediary step, the *Backend Application* would have to encrypt all its *Backend Order IDs* with the IV that was used to encrypt the *Backend Order ID* in the record which the law enforcement authority obtained from the *Frontend Company*. We must keep in mind that the law enforcement authority would not obtain this information due to using *Private Set Intersection*. However, for the *Private Set Intersection* to work, randomised padding, among other things, must not be used (see Section 6.5.2). Hence, it would likely be more purposeful to change the implementation described in Section 5.6.9 insofar as to additionally save a copy of

the encrypted *Frontend/Backend Order IDs* – which are stored in the respective "remote" application database – also in the "local" application database.[72]

In every case, subsequently, a *Private Set Intersection* and then *Oblivious Transfer* could be performed to provide the law enforcement authority with the required information.

It should be noted that this process could be performed also in the opposite direction – i.e. starting with the *Backend Company* – if, for example, the law enforcement authority – instead of customer-related information – initially had information about a certain purchase that was made by the *Backend Company* that it wanted to investigate. Finally, it must also be stressed that the whole process is not very efficient, since, for example, the information stored in the databases would presumably have to be preprocessed (e.g. decrypted) for *Private Set Intersection* and *Oblivious Transfer* to be carried out.

[72] The ID should be double encrypted in the local database so that under normal circumstance it cannot be matched easily with the respective bit representation that was sent to the other application.

6. Evaluation

We evaluate our proposed architecture and prototype for an *All-Steps Anonymous Purchase Platform (ASAPP),* which are based on the concept of *Separation of Data,* using the criteria *Privacy & Anonymity*, *Security* and *Deployability* which were largely based on the criteria and properties presented in [6]. Please note that we expanded the first criterion, which only reads *Privacy* in [6], to *Privacy & Anonymity* as these are related but not synonymous terms.

The first two criteria *Privacy & Anonymity* and *Security* require the description of a threat model to be evaluated against. This description is given in the following section. The subsequent sections 6.2, 6.3, and 6.4 cover the actual evaluation before we describe limitations of our architecture and prototype in Section 6.5.

6.1. Threat Model

Our threat model considers the parties who are involved in the purchase process (see Figure 5) and also potential external adversaries, i.e. adversaries that are not involved in the purchase process itself. In particular, we assume that:

The *GhostBuy Platform (Prototype)* operators, i.e. the *Frontend Company* and the *Backend Company*, are *honest but curious*: The platform itself performs its processes correctly but it is possible that employees at the companies try to obtain information which they are not eligible to obtain and/or draw conclusions from the information they are processing or which they additionally obtained. In addition, it is possible that employees from both companies collude in order to link the respective pieces of information that they obtained.

The involved online shops / e-commerce platforms, for example eBay or Amazon, are also *honest but curious*: They correctly fulfill their duties but use the information they obtain about any activities on their platforms – for example about product searches and purchases made by customers of *GhostBuy* (note, that means *through* the *GhostBuy Platform* on the respective e-commerce platform) – as they also use and process the information from their other customers. They will also try to use other information they have (e.g. from tracking cookies that they place in their customers' browsers) to link any activities performed by *GhostBuy* on their platforms to their customers. Also, individual employees might try to obtain more information by colluding with employees of the other companies involved in the purchase process.

The involved shippers, i.e. *Shipper 1* who ships packages from the online shop / e-commerce platform to the *Backend Company* and *Shipper 2* who ships packages from the *Frontend Company* to the *Customer (C),* are *honest but curious* as well. Please note that in between those two shipments the packages are processed and handed over from the *Backend Company* to the *Frontend Company* as described in Section 5.7. The shippers correctly fulfill their duties, but they will try to use the information that they obtain the same way as they use information about their other customers. Individual employees may try to collude with employees of the other companies involved in the process.

The involved banks are *honest but curious*, too. In particular, the following banks are involved: Banks, that handle the payments from customers to the Frontend Company (customers' banks and *Frontend Company's* bank(s), called *Bank 1* in the following), banks which transfer the necessary funds from the Frontend Company to the Backend Company (*Frontend Company's* bank(s) / *Bank 1* and *Backend Company`s* bank(s), called *Bank* 2 in the following) and banks which handle the payment from the *Backend Company* to the e-commerce platforms (*Backend Company`s* bank(s) / *Bank 2* and e-commerce platform's bank(s)). Like the other involved companies, the banks will correctly perform their assigned tasks, but they will use the information they obtain as they also use the information from their other customers. Individual bank employees may try to collude with employees of the other companies involved in the process.

Please note that we neither include the customers' banks / credit card companies nor the e-commerce platforms' banks into our further discussion since these companies obtain the same or less information than the other companies involved in the process (in line with simplified architecture illustrated Figure 5). That is, we only consider *Bank 1*, which receives payments from the customer and forwards them to *Bank 2* (potentially aggregated), which then performs the payments to the online shops / e-commerce platforms.

Potential external adversaries are at least *curious* but may also be *malicious*, i.e. they can try at least to eavesdrop on the communication between the involved parties or they can even try to manipulate it. Potential adversaries therefore can be rather weak (e.g. a customer's curious spouse that uses the same computer/device) but also rather strong (e.g. hackers trying to tap into the communication between involved parties – including the customers themselves

– also by manipulation,). Adversaries can either work offline, which means they have access to the customer's computer/device when the customer's session is expired / the session cookie has been deleted. Alternatively, adversaries can work online, i.e. while the session still is active, with or without access to the customer's computer/device.

All involved parties and potential external adversaries are *computationally bounded.*

6.2. Privacy & Anonymity

We begin the evaluation of our proposed architecture and the *GhostBuy Prototype* with the criterion *Privacy & Anonymit*y. However, as we described in Section 1.1.1, *anonymity* in online shopping is a technical measure to ensure *privacy,* as it means – with regards to the legal and practical reasons mentioned therein – that under *normal circumstances*, a consumer cannot be linked to purchases made on their behalf. Also, as we describe in the following, our proposed architecture and prototype can realize actual anonymity only for certain parts of the purchase process, respectively only regarding certain parties involved in the process due to the practical and legal reasons, but it still realizes privacy for the other parts / regarding the other parties. Consequently, we also extended the property "*Customer Anonymity"* (*"User Anonymity"* in [6]) that is evaluated in the next section to read *"Customer Privacy & Anonymity".*

6.2.1. Customer Privacy and Anonymity

We defined the property *Customer Privacy & Anonymity* as to be fulfilled, if:

> *No party involved in a transaction with the Backend Company (Bank 2, online shop, Shipper 1, the Backend Company itself…) and no third party may obtain any identifiable information of the customer on whose behalf the transaction was performed.*

It should be noted that our approach *can* fulfill this property only based on above definition, since when we defined our criteria, it was already foreseeable that for example the customer would have to specify personal and identifiable information for transactions with the *Frontend Company*, *Bank 1* (and the customer's credit card company) and that *Shipper 2* (delivering to the customer) will also obtain the customer's shipping address. Therefore, we also specified:

> *The platform architecture has to ensure that information obtained by the Frontend Company cannot be linked to information obtained by the Backend Company except for the processes controlled by the platform (e.g. there has to be a way to deliver goods to the customer, but the online shop must not be able to make a link to the customer).*

These definitions therefore are in line with Section 1.1.1, in which we argued:

> *"From the consumer's perspective staying anonymous under normal circumstances, especially not being linked to a certain purchase (of certain products), is the required property and the maximum possible approximation to actual anonymity with regards to the legal and practical reasons mentioned before."* (see Section 1.1.1)

1. To illustrate that our proposed architecture and prototype fulfills the property *Customer Privacy & Anonymity* as defined above, we created Table 5 that illustrates which data (rows) is known to which party/application (columns) involved in the purchase process. Please note that the column for "*Customer / Client Application"* does not need to be examined more closely as the *Client Application* is executed locally on the customer's device, which means that the information therein is not actually shared with anyone. Similarly, we will also not examine the "*Personal Details (for search term check)*" (top row) in more detail, as this information is created (entered) and used only within the *Client Application*.

Regarding the evaluation of the property *Customer Privacy & Anonymity* Table 5 shows:

- From the point of view of the online-shop-facing parties (i.e. the *Backend Company* and the parties it interacts with) the purchase process is performed completely anonymously, since these parties do not obtain identifiable, customer-related information, not even meta data such as the customer's IP address and tracking cookies. This can be seen from the fact that the cells in the intersection of the rows marked by the upper curly bracket on the left with the columns marked by the right curly bracket at the top are all green/unfilled – except for the cells within the row regarding browsing and purchase time which is discussed further below.

- From the point of view of the customer-facing parties (i.e. the *Frontend Company* and the parties it interacts with) the purchase process is not – but also cannot be – anonymous, since certain identifiable information about the customer must be shared with these parties. However, the customer's privacy is still maintained, since those parties only obtain the absolute minimum of information required to fulfill their duties, as can be seen from the scarcity of information that is shared with *Bank 1* and *Shipper 2*.

- Moreover, in general all parties either obtain identifiable, customer-related information or product-related information, but never both. This effectively means that they either know "who searched/purchased something" or "what someone searched/purchased", but they never know both. This can be seen by examining the upper, blue-bordered rectangle (1): Wherever there is customer-related information shared with parties (red/filled cells above horizontal dashed line) there never is product-related information shared with these parties at the same time (green/empty cells below horizontal dashed line) – and vice versa. However, this vertical data separation is not given for the browsing/purchase time and the *Order Cart Collection Total,* as discussed below.
- In addition, all involved parties can in general neither link the customer to a certain purchase nor link a purchase to a certain customer – except within the processes controlled by the *GhostBuy Platform* (e.g. package handover). This can be seen from the fact that within the outer, blue-bordered rectangle (1+2) there is no row that has red/filled cells on both sides of the vertical dashed line[73] – which means that no information that is obtained by customer-facing parties can also be obtained by online-shop-facing parties – and vice versa. However, this horizontal data separation is also not given for the browsing/purchase time and the *Order Cart Collection Total.*

[73] This line depicts the *Separation of Data* between the *Frontend Company (*and the parties it interacts with) and the *Backend Company (*and the parties it interacts with).

6. Evaluation

Table 5: Data knowledge by party/application

			Customer-facing parties (*Frontend Company* & parties it interacts with)					Online-Shop-facing parties (*Backend Company* & parties it interacts with)			
		Party / Application	*Customer*	Bank 1 (*Customer* to *Frontend Company* to *Backend Company*)	Shipper 2 (*Frontend Company* to *Customer*)	*GhostBuy Platform (Prototype)*			Bank 2 (*Frontend Company* to *Backend Company* to Online Shop(s))	Online Shop(s) / E-Commerce Platform(s)	Shipper 1 (Online Shop(s) to Backend)
						Frontend Company		*Backend Company*			
	Data		Client Application			Portal Application	Frontend Application	Backend Application			
Identifiable customer-related information	Personal Details (for search term check)	Given Name(s), Last Name, E-Mail, Address, City, Postal Code, Province	◘ (optional)	☐	☐	☐	☐	☐	☐	☐	☐
	Meta Data	Browsing Time, Purchase Time	■	◘ (purchase time)	◘ (approximate purchase time)	■	◘ (purchase time)	■	◘ (purchase time)	■	◘ (approximate purchase time)
		IP Address, Operating System, Browser, Tracking	■ (unused)	☐	☐	■ (no tracking)	☐	☐	☐	☐	☐
	Account Details	Account Name	■	☐	☐	☐ (hashed)	☐ (hashed)	☐	☐	☐	☐
		Password	■	☐	☐	☐ (hashed)	☐ (hashed)	☐	☐	☐	☐
		First Name, Last Name, E-Mail (optional)	■	☐	☐	■	■	☐	☐	☐	☐
	Shipping Details	Full Name, Address, City, Postal Code, Province	■	☐	■	■	■	☐	☐	☐	☐
	Payment/ Credit Card Details	Network, Currency, Holder, Number, Valid Through, Security Code	■	■	☐	■	■	☐	☐	☐	☐
Product-related information	Product Searches	Shop, Category, Price Range, Sorting, Search Terms, Displayed Products / Product Details	■	☐	☐	☐ (encrypted with backend/client key 6-B/C)		■	☐	■ (for searches in respective shop)	☐
		Shopping Cart Content	■	☐	☐			■	☐	☐	☐
	Order Details	Order Cart(s) Shop(s)	■	☐	☐	☐ (encrypted with backend/client key 6-B/C)		■	■	◘ (for respective cart)	◘ (for respective cart)
		Order Cart(s) Total(s)	■	☐	☐			■	■	◘ (for respective cart)	☐
		Order Cart(s) Product(s): Details, Quantities, Prices, Shipping, Fees, Tax	■	☐	☐			■	☐	◘ (for respective cart)	☐ (only "own" shipping costs)
Order reference information		Backend Order ID	☐	☐	☐	☐	☐ (encrypted with backend key 7-B)	■	☐	☐	☐
		Backend "Tracking ID"	☐	☐	☐	☐	☐	■	☐	■	■
		Frontend Order ID	☐	☐	☐	■	■	☐ (encrypted with frontend key 9-F)	☐	☐	☐
		Frontend "Tracking ID"	■	☐	■	■	■	☐	☐	☐	☐
Product-related information		Order Cart Collection Total (incl. platform fees & taxes)	■	■	☐	■	■	■	■	◘ (calculable)	☐

Upper rectangle (1)

Lower rectangle (2)

■ = data is known to party / application
◘ = data is at least partially known to party / application
☐ = data is unknown to party / application

So, as we can see in Table 5, two types of information *are* actually obtained by parties on both sides of the vertical dashed line – i.e. by customer facing parties and online-shop-facing parties – namely the customer's product browsing and purchase times (row above outer blue-bordered rectangle) and the *Order Cart Collection Total* (row below outer blue-bordered rectangle). To recall, this is a violation of the horizontal data separation. Also, since time information is therefore also obtained by parties who otherwise only process product-related information, and the *Order Cart Collection Total* is therefore also obtained by parties who otherwise only process customer-related information, this is at the same time a violation of the vertical data separation.

Regarding the product browsing and purchase times it is obvious that we cannot avoid this type of information being obtained by all parties involved, since our *GhostBuy Prototype* realizes real-time product searches/purchases and since common online shops often ship the products in a timely fashion. Regarding the *Order Cart Collection Total,* we also cannot completely avoid sharing this information between the *Frontend Company* and the *Backend Company*, since for example funds must be transferred between them. Yet, it would be possible to improve our prototype implementation for example by making the *Frontend Company* add its own fees and taxes to the total amount that is specified (and shared) by the *Backend Company.* Additionally, the totals could be disguised at least to a certain degree, for example by rounding or slightly randomizing the fee amounts. Also, funds could be aggregated before being transferred between the banks of the *Frontend Company (Bank 1)* and the *Backend Company (Bank 2)*, for example on a daily basis. These measures would make it even harder for (employees of) the parties involved to link customers and purchases based on the respective payments.

This shows that obviously the risk of correlation attacks by (employees of) the parties involved cannot be completely mitigated. Nevertheless, the larger the number of purchases that are made within a certain period of time and the more customers, banks, shippers and online shops can potentially be involved in different purchases, the more unlikely it is that these attacks will succeed: This is true since a larger number of purchases will for example increase the number of similar-priced orders (anonymity set) and a larger number of potentially involved parties makes it harder for attackers to match correlating data. These effects can be artificially

amplified, not only by disguising order prices, but for example also by processing / handing over packages in batches, by using alternating shippers for package delivery to the customer and by incorporating several banks which alternately process customer payments. In addition, organizational aspects can help mitigate the risk of correlation attacks as well: For example, employees of the *Backend Company* should only obtain the minimum necessary amount of information to fulfill their duties: For instance, even though the *Backend Company* obtains data about the ordered products, this information needs not get known to individual employees (see Section 5.7) unless, for example, they handle package returns.

Also, it should be noted that customers of *GhostBuy* can be identified by (employees of) *Bank 1* and Shipper 2 as just that – customers of *GhostBuy* – whenever they complete a purchase, as *GhostBuy* is the recipient of the customers' payments and the sender of the packages sent to the customers.

However, to put these downsides and limitations into perspective: The goals of *GhostBuy* regarding privacy and anonymity are to make the customers anonymous from the perspective of the online shop(s), *Bank 2*, *Shipper 1* and the *Backend Company* while at the same time preserving the customers' privacy regarding their exact product searches and purchases from the perspective of all other parties involved, including the *Frontend Company*. As this section shows, these goals have been achieved with the *GhostBuy* architecture and prototype. This is true especially if we assume that the involved GhostBuy-external parties would not put the necessary effort into linking purchases to customers by finding correlating data across several parties. We argue that this assumption is reasonable, as anything else would be illegal and not worth neither the risk nor the effort if the GhostBuy-external parties' only intention was to get a better understanding of their customers[74]. This rationale is also consistent with our threat model's assumption that these parties are *honest but curious*. Adversaries who are not employees of any party involved in the purchase process on the other hand would have to put even more effort into correlation attacks, since they would first have to (illegally) obtain the data from the parties involved.

[74] Also known as "Big Data".

With regards to the *GhostBuy Platform* itself – and the possibility that employees of the *Frontend/Backend Companies* could try to mount joint correlation attacks, potentially also incorporating GhostBuy-external parties, we note that the platform's GUIs for employees should only yield the minimum necessary information (see Section 5.8) and that the platform's applications also would be monitored for these types of activities (see Section 6.3.3).

Finally, we want to point out that even in the case of criminal investigations the *GhostBuy Platform* would potentially be able to better protect its customers' privacy and anonymity than common online shops / e-commerce websites as outlined in Section 5.10.

6.2.2. Transaction Unlinkability

We defined the property *Transaction Unlinkability* as fulfilled, if the following is given:

> *No party involved in a transaction with the Backend company including the Backend Company itself and no third party may be able to attribute two transactions to having been performed on behalf of the same (unknown) customer.*

Regarding this definition, we would like to point out that this only applies if a customer did not give consent for linking the transactions carried out on their behalf. Please note that this constraint was given in the original dissertation from which we adapted this property definition [6, p. 27].

Table 6: Transaction unlinkability by party

	Transaction unlinkability regarding customer transactions			
	Customer-facing, GhostBuy-external parties (parties that interact with Frontend Company A)	**GhostBuy Platform**		**Online-Shop-facing, GhostBuy-external parties** (parties that interact with Backend Company B)
Term "transaction" is defined as		***Frontend Company***	***Backend Company***	
Pre-purchase transaction performed on behalf of customer during same session: e.g. Product Search, Shopping Cart Modification	**yes** (parties are not involved in these transactions)	**no** (transaction is linkable, its ***action type*** is known but its ***content*** is encrypted)	**no** (**virtual** session state is maintained here, but transactions are not monitored)	**yes** (all transactions performed via respective API on behalf of Backend Company B)
Pre-purchase transaction performed on behalf of customer in different sessions (without signing-in): e.g. Product Search, Shopping Cart Modification	**yes** (parties are not involved in these transactions)	**yes** (in each session *Initial Cryptographic Setup* is built with random settings)	**yes** (in each session *Initial Cryptographic Setup* is built with random settings)	**yes** (all transactions performed via respective API on behalf of Backend Company B)
Purchase performed on behalf of customer in different sessions	**no** (customer can be identified e.g. by his shipping address or payment information)	**no** (even with new account customer could be identified e.g. by address)	**yes** (in each session *Initial Cryptographic Setup* is built with random settings)	**yes** (each purchase performed via respective API on behalf of Backend Company B)

We created Table 6 to provide an overview of which transactions (rows) can be linked by which party involved in the *GhostBuy Platform's* purchase process (columns). Hereby we

differentiated between transactions that are performed during the same session (upper row) and transactions that are performed in different sessions (lower two rows). First, Table 6 shows that the *Frontend Company* and the parties it interacts with can attribute two transactions to the same, unknown customer when shipping and payment information were specified for a purchase (bottom row, left of dashed line). However, even ahead of the prototype implementation we foresaw that *Transaction Unlinkability* would not be given for transactions with the *Frontend Company* and the parties it interacts with for exactly this reason. In addition, it is obvious and also unavoidable that while the customer's session remains active (or if they sign in during different sessions), the *Frontend Company* can also link the corresponding transactions to the customer.

So, our actual goal was to achieve *Transaction Unlinkability* for the *Backend Company* and the parties it interacts with, and – as one can see from the two columns right of the dashed line in Table 6 – the *GhostBuy Platform* accomplishes that goal except for the case that transactions take place within one session: The *Backend Company* could attribute such transactions to the same, unknown customer, since it maintains a virtual session state[75] for the customer. However, we argue that the customer implicitly gave their consent for this, since otherwise handling a shopping cart for the customer would not be possible. Of course, unlike common practice of online shops, the *Backend Company* should also not / would not analyse transaction data for marketing purposes or the like – after all, its business idea is to protect the privacy and anonymity of its customers when shopping online. Additionally, its systems should also not allow employees to access transaction data outside of well-defined workflows as for example the workflow described in Section 5.7. It should be kept in mind that although two transactions can be attributed to the same customer, this customer still remains unknown to the *Backend Company*.

Furthermore, the bottom two rows right of the dashed line in Table 6 show that when the customer's session ends – which also happens when a purchase has been completed or the customer signs out – subsequent transactions are performed in a newly created session and can no longer be linked to the same customer neither by the *Backend Company* nor the parties

[75] Within the web session of the Frontend Company's Portal Application, based on *Backend Session Key*.

it interacts with. It also seems reasonable to require that customer transactions be unlinkable only if they take place in different sessions respectively if they are completed purchases, since transactions that take place in the same session can by definition be linked to the same customer.

In conclusion, the *GhostBuy Platform (Prototype)* technically provides *Transaction Unlinkability* for all transactions performed by the *Backend Company* and the parties it interacts with, except for transactions performed within the same (virtual) session – which should either not be considered here or to which – we argue – the customer gave their implicit consent since handling a session / handling a shopping cart is only possible with linkable transactions. Thus, the property *Transaction Unlinkability* is fulfilled. Yet, it should be noted that successful correlation attacks which for example could be based on time information (see Section 6.2.1) could render transactions linkable, too.

Finally, we want to point out that the *GhostBuy Prototype* provides *Transaction Unlinkability* also when a customer reviews multiple earlier purchases during the same session as explained in Section 5.6.9.

6.3. Security

We continue the evaluation of our proposed architecture and the *GhostBuy Prototype* with the criterion *Security.* Similar to the definition of *Security* in [6] this criterion comprises the properties *Functional Correctness*[76], *Fairness*, *Accountability*, and *Mis-Authentication Resistance*. We added the properties *Investigatability* and *Cryptographic Attack Resistance* which were not given in [6].

6.3.1. Functional Correctness

At first sight, the property *Functional Correctness* seems to be of rather academic nature as it requires the following:

> *The platform is able to achieve its goals (for example ordering and delivery of goods to the customer) if all involved parties are honest.*

[76] To avoid confusion with the property ***Basic*** *Functionality* of the criterion *Deployability (*see Section 6.4.1), we named the property *Functional Correctness* (instead of *Functionality*) – as it was similarly named in [6].

Yet, although this requirement somehow seems to be superfluous to state – since a platform / any application that does not achieve its goals is simply useless – conversely, it is important to note that the *GhostBuy Platform Prototype* and any other application based on our architecture do *not* achieve their goals if *not* all parties involved in the purchase process are honest. For example, if *Shipper 2* would x-ray packages before delivering them to the customer the platform would not achieve its goal of preserving the customer's privacy. Or, if the online shop would not put the correct products into the package that is shipped to the *Backend Company,* the customer will not receive the products they purchased, since by default the *Backend Company* may not review the products in order to preserve the customer's privacy (see Section 5.7).

In conclusion, the property *Functional Correctness* is fulfilled since the platform does achieve its goals if all involved parties are honest – but it does not if not all of them are honest. However, a dishonest party can be detected, if the property *Accountability* is fulfilled (see Section 6.3.3).

6.3.2. Fairness

For the property *Fairness* we defined the following:

> *The platform fulfills the property Fairness if it ensures that all parties involved receive their respective benefit – for example the online shop its payment and the customer their package – if and only if the respective party correctly completed its duty.*

While this definition abstracts the ideal case of an in-person purchase where the principle "money for goods" is commonplace, in reality many online shops and most shippers that we know of rather follow the principle "goods/service for money", i.e. they have to be paid first and will perform their duties afterwards. Since the *GhostBuy Platform* transparently integrates into existing business processes (see Section 6.4.5) we cannot change this principle, at least not for the online shop and its respective shipping service (*Shipper 1*). We are also not convinced that there would be a *Shipper 2* that would allow payment for its service to be made only after successful delivery.

Therefore, we argue that it is reasonable to enhance the definition so that the property is also fulfilled if a certain party receives *and keeps* its benefit if and only if it correctly completed its duty. That is, a party may be paid before it has fulfilled its duty, but if it does not fulfil it correctly, there must be a way to reclaim the payment or to enforce correct fulfilment.

Table 7: Fairness by party

Purchase Process Step	Party	Duty	Benefit	*Fairness achievable by*			Property *Fairness* can be fulfilled
				Duty fulfilment before benefit	Duty fulfilment enforceable	Prepayment reclaimable	
1	**Customer**	**Payment to Frontend Company**	**Products delivered**	✓	n/a		✓
2	**Bank 1**	**Customer payment processing / transfer of funds**	**Fees paid**	✓	n/a		✓
3 & 9	**Frontend Company**	**Purchase brokerage, package handover** from Backend Company **and shipping of products** with *Shipper 2* to *Customer*	**Payment received** from *Customer*	✗ (only brokerage)	✓ (by customer)		✓
4	**Bank 2**	**Transfer of funds / processing of payment to online shop**	**Fees paid**	✓	n/a		✓
5 & 8	**Backend Company**	**Purchase brokerage, package handling and handover** to Frontend Company	**Payment received** from *Frontend Company*	✗ (only brokerage)	✓ (by Frontend Company)		✓
6	**Online Shop**	**Shipping products** with *Shipper 1* to *Backend Company*	**Payment received** from *Backend Company*	✗	✓ (by Backend Company)		✓
7	**Shipper 1**	**Delivering the products shipped** by Online Shop to Backend Company	**Payment received** from Online Shop	✗	✓ (by Online Shop)		✓
10	**Shipper 2**	**Delivering the products shipped** by *Frontend Company* to *Customer*	**Payment received** from Frontend Company	✗	✓ (by Frontend Company)		✓

In Table 7 we provide an overview of the fulfilment of the property *Fairness* by each party involved in the purchase process. The table shows that each party fulfils the property, either because it receives its benefit only after having performed its duties (*Customer, Bank 1* and *Bank 2*) or because fulfilment can be enforced, or the prepaid amount can be reclaimed (*Frontend Company*, *Backend Company*, *Online Shop*, *Shipper 1* and *Shipper 2*).

It must be noted that the exchange of duties and benefits shown in Table 7 is not actually triggered in the *GhostBuy Prototype* but rather simulated (customer payment, order at online shop) or just conceptually described. However, if the entire purchase process were carried out as outlined, the property *Fairness* would be fulfilled by the *GhostBuy Platform*.

6.3.3. Accountability

The property Accountability is fulfilled if the platform's protocols and processes ensure that dishonest (or incapable) parties are detected and identified.

The challenge in fulfilling this property is that the *GhostBuy Platform* transparently integrates into existing processes and protocols of third-party companies so that firstly, ensuring

accountability is not entirely in the platform's hands and secondly, this cannot be done in all cases purely based on – cryptographic – protocols. Instead, detecting fraud or incapability would often be entirely in the hands of a GhostBuy-external party, although an investigation might still be triggered by the platform: For example, if the *Customer* complained that they did not receive the correct products, the *GhostBuy Platform* might have to request the online shop (and potentially *Shipper 1)* to investigate the matter, if it can rule out that the issue occurred within the platform itself or with *Shipper 2* (see Section 6.3.3). In most cases, the issue / accountable party should be detectable. If, however, the *Customer*– for example – claims that products were missing and the online shop insists that it did send the correct products, the platform would not be able to establish with certainty that the *Customer* had misappropriated the missing products – unless the online shop would provide an actual proof regarding the products they shipped (and *Shipper 1* could also prove that the products were not removed by them). In this case the *Backend Company* would try to resolve the issue with the respective online shop which will presumably be interested in reaching an agreement. It is however possible that in some cases no agreement can be found and e.g. the customer would have to be refunded even though the online shop did not refund the *Backend Company.* Cases like these should be seen as a business risk of the *GhostBuy Platform.* Nonetheless, the *Frontend Company* might want to record incidents like these (without details) to identify whether a particular customer is likely to be dishonest.

The point is, as long as the *GhostBuy Platform* shall transparently integrate into existing processes of common online shops and shippers, the property *Accountability* cannot be fulfilled to 100 percent – but in many cases, especially those who are in direct control of the GhostBuy Platform, *Accountability* will be given. Considering this, we note that there is a trade-off between the platform's *Integration Transparency* (see Section 6.4.5) and its *Accountability*.

Table 8 gives an overview of the accountability by party and duty. We want to emphasize that the rightmost column illustrates the accountability for the respective party and duty that *potentially* would be achievable in a complete implementation of a GhostBuy-like platform but not the accountability that the *GhostBuy Prototype* actually provides. For the readers

reference, the penultimate two columns on the right state which means to ensure accountability we have implemented in the *GhostBuy Prototype.*

Table 8: Accountability by party and duty

Purchase Process Step	Party	Accountable for	Accountability achievable by	Implementation in prototype / Process Description	Property *Accountabilty* can be fulfilled
1	Customer	**Payment to *Frontend Company***	Confirmation of payment (*Bank 1*)	**Yes (simulated)**	✓
11		**Accepting delivered package**	Package verification (*Customer*)	**Description in Section 5.10**	✓ / ✗
2	Bank 1	**Customer payment processing / transfer of funds**	Accounting (*Bank 1*)	n/a	✓
3	Frontend Company	**Purchase brokerage**	Records order authentication hashes of *Customer* and *Backend Company*	**Yes**	✓
9		**Handover of package** *from Backend Company*	Package handover is logged	**Yes**	✓
			(Employee should be authenticated)	**No**	
		Shipping of products with *Shipper 2* to *Customer*	Shipping Confirmation (Shipper 2)	n/a	✓
4	Bank 2	**Transfer of funds / processing of payment to online shop**	Accounting (Bank 2)	n/a	✓
5	Backend Company	**Purchase brokerage**	Authentication hash and encrypted order are recorded / saved by Frontend Company & order is logged	**Yes**	✓
			Order confirmation (*Online Shop*)	n/a	
8		**Package handling** (receive & repackage)	Confirmation of receipt (Shipper 1), repackaging minuted electronically	n/a	✓
		Handover of package to *Frontend Company*	Package handover is logged	**Yes**	✓
			(Employee should be authenticated)	**No**	✓
6	Online Shop	**Shipping products** with *Shipper 1* to *Backend Company*	Accounting (*Online Shop*) & Shipping Confirmation (*Shipper 1*)	n/a	✓ / ✗
7	Shipper 1	**Delivering the products shipped** by *Online Shop* to *Backend Company*	Shipping Confirmation, Confirmation of receipt through *Backend Company* , Accounting (*Shipper 1*)	n/a	✓ / ✗
10	Shipper 2	**Delivering the package shipped** by *Frontend Company* to *Customer*	Shipping Confirmation, Confirmation of receipt through *Customer* , Accounting (Shipper 2)	n/a	✓

6.3.4. Investigatability

The property Investigatability requires that the platform be able to adhere to the legal requirement of providing law enforcement agencies with information that they might request as part of a criminal investigation.

For example, the platform must be able to establish which transactions at the *Frontend Company* are linked with certain transactions performed by the *Backend Company* and vice versa. Similarly, law enforcement agencies might also ask which orders have been performed on behalf of a certain customer – in which case the customer's transactions would have to be obtained from the *Frontend Company* first and afterwards the link to the respective transactions – i.e. orders – at the *Backend Company* would have to be made. It is obvious that the legal requirement to be able to satisfy such inquiries conflicts with the requirement that under normal circumstances it must not be possible to link transactions outside of the platforms processes.

In Section 5.7 we show that the *GhostBuy Platform* can link transactions performed at the *Backend Company* to transactions performed at the *Frontend Company*. We recall that this process is based on that the *Backend Company* stored a reference ID (the *Frontend Order ID*, encrypted by the *Frontend Application*) with the Backend Order DBO, which it can retrieve based on the *Backend Tracking ID* that identifies a *Backend Order* DBO – of course, any other distinct order-related information would be equally good. Since a similar reference ID is also stored with the *Frontend Order DBO* this process also could be implemented for the opposite direction.

In conclusion, this means that the property *Investigatability* is fulfilled as transactions can be linked to the customer at the Frontend Company and transactions can be linked between the *Frontend Company* and the *Backend Company* in both directions.

In Section 5.10 we describe how law enforcement agencies could obtain the required information while preserving *Customer Privacy and Anonymity* as much as possible.

6.3.5. Cryptographic Attack Resistance

The security property *Cryptographic Attack Resistance* was not defined ahead of the implementation as it was not given in [6], either. However, only evaluating the *Mis-Authentication Resistance* (next section) would not properly consider if the platform is secure in itself. That is, if the cryptographic algorithms used by the platform have a lower security level than its authentication mechanisms the whole platform's security is limited to this security level. Therefore, we analyze the platform's strength and weaknesses against cryptographic attacks in the following:

Regarding our choice of cryptographic algorithms, we want to recall that we had settled on the *Web Crypto API* for implementing the *Client Application*. Therefore, we were limited to use the best algorithms that are supported by the *Web Crypto API* for building the JavaScript-based *Client Application* and the Java-based applications it directly and indirectly interacts with – even though these would have supported other, stronger algorithms (see Section 4). For the *GhostBuy Prototype*, we needed to use/choose an asymmetric encryption algorithm or dedicated key agreement algorithm for the key agreement between the *Client Application* and the *Backend Application*, a symmetric encryption algorithm for all other encryption purposes,

a hash function and – as we realized later – a key derivation algorithm for generating the *Client-side Secret Key (5-C)* which is based on the customer's password. The key derivation function is also used for hashing account names and passwords. Table 14 in Appendix P gives an overview of the cryptographic algorithm implementations and configurations in the *GhostBuy Prototype.* Below, we will briefly explain our choices for the required algorithms:

We chose to use RSA4096 as the asymmetric encryption algorithm, since when we started the implementation it was the strongest asymmetric algorithm supported by all browsers that we intended to support/test at that time. The browser selection included the "legacy" Microsoft Edge Browser (Spartan)[77] as the new Microsoft Edge (Anaheim) had just been introduced at that time. However, Edge Legacy's *Web Crypto API* implementation did not support *Elliptic Curve Diffie-Hellman (ECDH)*, which would have otherwise been a good alternative regarding the choice of an asymmetric encryption algorithm, respectively key agreement algorithm. Unfortunately, we later had to stop supporting the legacy Edge Browser, since it suddenly stopped executing our custom JavaScript modules for no apparent reason (no error message). Also, it did not support PBKDF2 (see below). However, we kept using RSA4096 since this part of the implementation was already completed at that point of time – and time was scarce.

For symmetric encryption we chose AES256-GCM, which was the strongest algorithm supported by the *Web Crypto API* and it was also implemented in all browsers. In addition to providing authenticated encryption with 256-bit security, it also allows for adding unencrypted *Additional Authenticated Data (AAD)*.[59] We used additional authenticated data to ensure that decryptions cannot be taken out of context. For example, we assigned a unique ID to sessions within the *Portal Application* (not the session ID, since this changes) so that data from one session cannot be decrypted within the context of another session.

We chose the SHA-256 hash algorithm for creating salted hashes in the *Client Application*, more precisely for generating the URL-ID, as described in Section 5.6.6. The *Web Crypto API* also supports SHA-384 and SHA-512 but we preferred to use SHA-256 due to its smaller outputs – regarding their use as URL-IDs. It should be noted that while SHA-256 only

[77] https://en.wikipedia.org/wiki/Microsoft_Edge

provides 128-bit collision resistance strength by default (see [60, Sec. 4.2]), our particular implementation is 256-bit collision resistant because we use a secret 256-bit salt that is consequently also encrypted with AES256-GCM when being saved to the *IndexedDB*.

Finally, we chose the *Password-Based Key Derivation Function 2 (PBKDF2)* with SHA-512 as hash function to derive the *Client-side Secret Key (5-C)* from the customer's password (see Section 5.6.7), to (re-)hash the password during authentication (see next section) and also to hash the *Order Cart Collection* (see Sections 5.6.8 & 5.6.9). In fact, PBKDF2 is the only key derivation algorithm supported by all *Web Crypto API* implementations of the browsers that the *GhostBuy Prototype* supports, but it is also well-studied and provides 256-bit collision resistance strength when used with SHA-512. [60, Sec. 4.2] In addition, we in general use it with a secret 256-bit salt that is stored encrypted with AES256-GCM when being saved to the *Frontend Application* database and *Backend Application* database (except for non security relevant account name hashes).

Figure 37 provides an overview of the cryptographic algorithms that are used in the *GhostBuy Prototype*, of where the respective encrypted content is stored and from where to where it is being transmitted. The most important observations that can be made are the following:

- A completely external adversary, i.e. an adversary who tries to eavesdrop on the communication taking place between the parties involved but who does not have direct access to a system of one of those parties, would need to break the encryption of the https-connections. Therefore, the platform provides at least 256-bit security against such an attacker.

- Even if a completely external adversary was able to eavesdrop on https connections, the only sensitive data which could be directly exposed to them would be account data (for credentials – hashes only, see Table 5) and the part of the order data that contains shipping information and payment information. Yet, if the attacker is able to mimic the *Client Application's* protocols – e.g. by locally running a *Client Application* with manipulated code, the obtained account data (credential hashes) could likely be used to mount an impersonation attack for the respective account(s). While such an attack could be carried out, for example, to obtain information about a specific customer, it

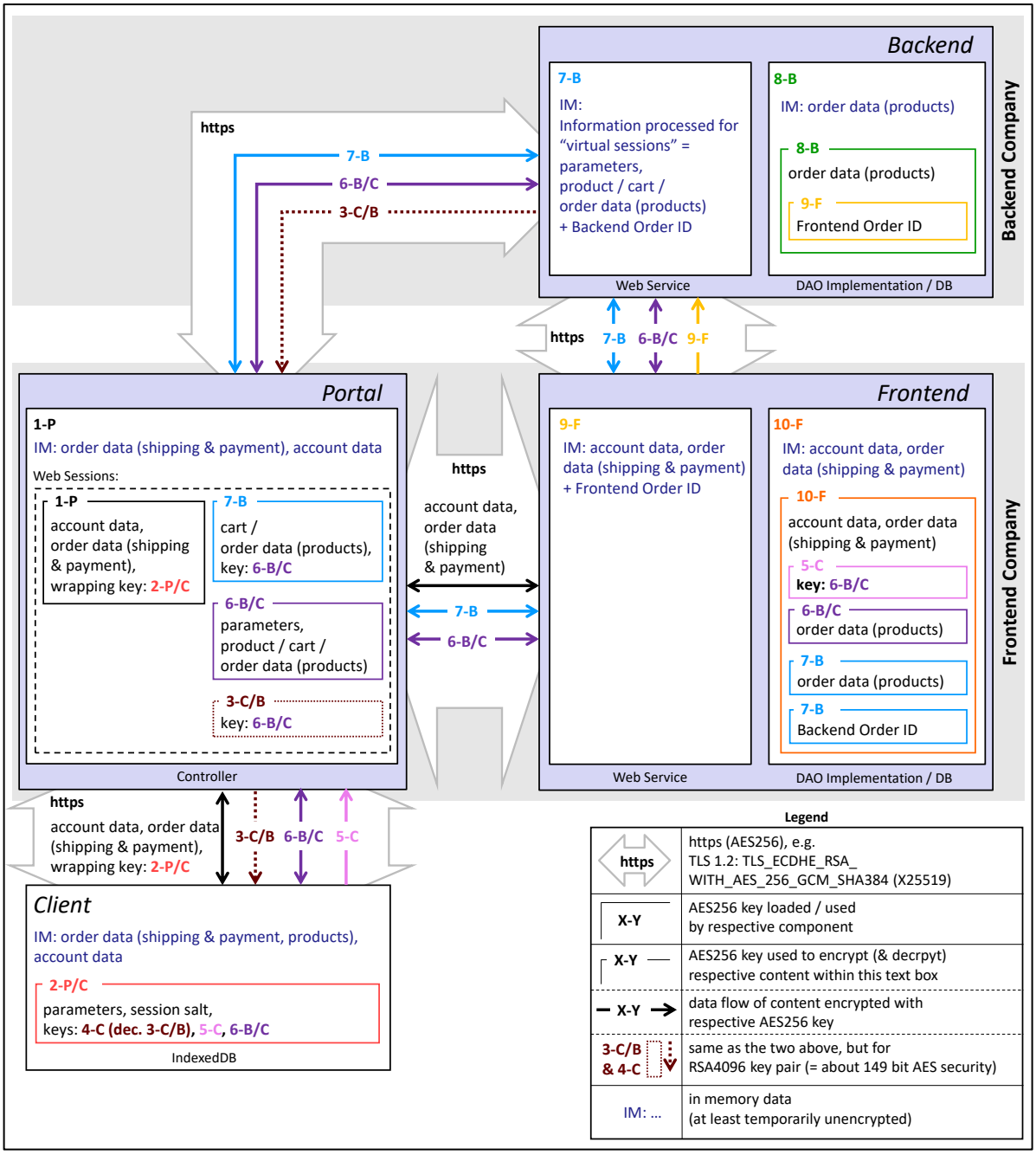

Figure 37: Overview of cryptographic algorithms and encrypted content used in GhostBuy Prototype

would not be efficient on a large scale unless the attacker was able to eavesdrop on multiple https connections (of different customers).

- An offline attacker, who (only) has access to data being stored by any party involved, needs to break AES256 encryption in order to obtain any sensitive information. Hereby, the presumably most interesting target would be the *Frontend Application`s* database since it stores all information of all customers. However, the product-related

part of the data is encrypted twice with AES256 (since it was encrypted by the *Backend Application* first).

- An online attacker who has direct access to a system of one of the parties involved would most easily obtain information by targeting a customer's device: Within the *Client Application*, all relevant information for that customer is decrypted and even if the attacker did not have physical access to the system (such as a spouse) they could obtain it by accessing the browser's rendering engine, for example. Also, the customer's credentials could be recorded for later use by the attacker. This means that the platform itself does not provide protection against such an attacker. However, this type of attack would only compromise the privacy and anonymity of a single customer and the attacker would need to gain access to the customer's device in the first place.

- If alternatively, an online attacker would seek to target (only) one of the other applications, the presumably most interesting target for them would be the *Portal Application* for the following two reasons: First, the customer-related information can be obtained in the *Portal Application*, including credential hashes sent by the Client Application, which could be used for impersonation attacks. Second, as an alternative to the impersonation attack, the attacker could try to break the RSA4096 encryption of the *Backend Session Key (6-B/C).* The RSA4096 encryption has only about 149-bit AES security[78] and therefore is considerably weaker than the AES256 encryption for which the *Backend Session Key (6-B/C)* is used. In conclusion, the platform does not provide protection against an active attacker who obtains credential hashes on the *Portal Application,* but it provides at least security equivalent to 149-bit AES if an impersonation attack cannot be mounted by the attacker. However, it must be noted that the attacker would need to gain "online" access to the *Portal Application* in order to perform such an attack (password hashes are rehashed and encrypted before they are saved to the database). The *Frontend Company* (and also the *Backend Company*)

[78] Respectively the key strength is approximately the same as for a symmetric key of that length. [66, p. 78]
Calculated with formula given in [66, p. 78]: $x = \frac{1.923 \times \sqrt[3]{L \times ln(2)} \times \sqrt[3]{[\ln(L \times ln(2))]^2} - 4.69}{ln(2)} \approx 149.341$,
where L = RSA keysize in bits (here, 4096).

should therefore seek to mitigate the risk of such an attack by for example implementing an intrusion detection system as well as restricting and monitoring the information access of its employees. Regarding an additional possible counter measure against credential-hash-based impersonation attacks please see the next section (Section 6.3.6).

- Finally, we argue that the usage of a *Client Wrapping Key (2-P/C)* that is stored and provided by the *Portal Application* not only binds the *Initial Cryptographic Setup* to the respective web session (see Section 5.5) but hereby also further mitigates the risk of so-called *Key Exchange API Attacks* on cryptographic keys used by the *Web Crypto API* that are described in [50] – at least if the attacker does not have online access to the web session or cannot access content of the *IndexedDB* (see Section 6.1). We would also like to point out that these attacks can only be rendered if the attacker is able to run malicious JavaScript code on the customer's device (see [50]) which in general should not be possible since the *Client Application* can only be provided by the *Portal Application* and also be run in a secure context.

Regarding the usage of RSA4096 for encryption of the *Backend Session Key (6-B/C)* by the Backend Application we want to recall that we had settled on the *Web Crypto API* for implementing the *Client Application*. Therefore, we were limited to use the best available algorithms that are supported by the *Web Crypto API* for building the Client Application and the applications it (indirectly) interacts with – even though these would have supported other, stronger algorithms (see Chapter 4). As mentioned earlier, this is a problem insofar as RSA4096 is equivalent to only about 149-bit AES security which therefore applies also to the protection of the *Backend Session Key (6-B/C)* that is encrypted herewith. RSA15360 would be suitable to protect an AES256 key but as of now is not supported by the *Web Crypto API*. [46, Sec. 5.6.1.1] To overcome this issue, we suggest using the ECDH key agreement algorithm instead of RSA4096, since it supports key agreement for 256-Bit AES keys[79] and is supported by all *Web Crypto API* implementations of the browsers that we tested. To recall, we had not originally chosen it, since it was not supported by the legacy Microsoft Edge

[79] When using named curve P-521 (521 bit key).[67, Sec. 5.6.1.1], [68, Sec. 13] Please note that NIST curves have received some criticism. [69]

Browser, which, however, we had to stop supporting for other reasons. It should be noted that ECDH as opposed to RSA requires that both the *Client Application* and the *Backend Application* create public key pairs for performing a key agreement, which means that more steps are necessary to create/agree on the *Backend Session Key (6-B/C)*. Like for the key distribution via RSA encryption, at least the client's key pair should be single-use only, so that the ECDH algorithm in fact would be ECDHE, where the last E stands for "ephemeral".

Also, in the future, the *Web Crypto API* will likely support more and stronger encryption and key agreement algorithms so that in the platform implementation RSA4096/ECDH could be replaced by a stronger asymmetric encryption algorithm or key agreement algorithm. This should also be seen in connection with future developments in quantum computing. While the AES256 algorithm will presumably still provide about 128-bit security against quantum computer brute force attacks, RSA encryption and ECDH will likely be broken by future quantum computers. [61] Accordingly, when quantum computers become powerful enough to actually perform the cryptanalysis algorithms described in [61], AES256 should and RSA4096/ECDH must be replaced by stronger algorithm variants respectively other, stronger algorithms, such as those currently being evaluated by NIST (see [62]).

In conclusion, our evaluation of the property *Cryptographic Attack Resistance* shows that our platform provides 256-bit security against external adversaries due to the platforms usage of accordingly configured https-connections. Attackers who have direct access to a system involved in the process will obtain sensitive data more easily but either only for a single customer (customer device) or only certain types of sensitive information (*Frontend Application* / *Backend Application)* since the other information is still protected by AES256 encryption. The platform's *Cryptographic Attack Resistance* could be further improved by replacing the RSA4096 encryption with a stronger algorithm (variant) or key agreement protocol. Also, the risk of attacks based on observing credential hashes can be further mitigated by using intrusion detection systems, monitoring and two-factor authentication (see next section).

6.3.6. Mis-Authentication Resistance

The security property *Mis-Authentication Resistance* refers to several aspects of the implementation:

> *The main focus of Mis-Authentication Resistance is on that no customer will be able to use the platform's services (for example order and receive goods) unless authorized to do so. The platform should also ensure that within the platform itself data access / information exchange is possible only when sender and recipient are authenticated.*

For example, a less rudimentary implementation of the package handover process (see Section 5.7) should require that employees authenticate before being able to access the respective input masks. Also, the web service interfaces should not only authenticate themselves to the web service clients (which we implemented via configuring self-signed certificates), but the clients should also authenticate themselves to the web service servers, for example by using client certificates or access tokens. However, a minimal level of client-side authentication is actually achieved by the fact that the web service clients need to specify secret knowledge to retrieve information from the web services, for example the Order ID of a to-be-retrieved Frontend Order DBO. As for the database servers, these are authenticated via self-signed certificates and only allow clients to connect if they authenticate via a password. This could be further improved by requiring client-side certificates, too. Regarding the authentication of customers who access the platform we must consider:

1. The explicit authentication of customers through the sign-up and sign-in processes.
2. The implicit authentication of the customers' web browsers within a web session, both before and after signing up or in.

Regarding the implementation of the sign-up and sign-in process we followed sections 5.1.1.1 and 5.1.1.2 of the *Digital Identity Guidelines on Authentication and Lifecycle Management* published by NIST [56, Secs. 5.1.1.1-5.1.1.2]. Table 9 on page 118 gives an overview of the requirements and recommendations specified therein and whether and how the *GhostBuy Prototype* is compliant with them. We would like to point out the following four key-takeaways:

1. We marked measures 2, 3 and 5 – all of them aiming at ensuring a sufficient password complexity – as "compliant" even though they are technically non-compliant. However, we argue that the respective requirements are not applicable for the password generator that we implemented for the prototype but that the respective measure's intention is fulfilled, nonetheless.

2. Measure 6 – imposing a limit on password generations – has not been specified by NIST, since it does not apply to common password authentication processes, where a user makes up a password. Should a password generator like ours be used in a real-world implementation, we suggest imposing such a limit to avoid that customers generate too many password phrases. Otherwise, a potential bias in their choice could increase password predictability.
3. Measure 11 – imposing a limit on failed authentication attempts – should be implemented in a real-word environment.
4. All other measures that are required or recommended by NIST were implemented in the prototype.[80]

We conclude that the *GhostBuy Prototype* is almost completely compliant with the NIST recommendations given in [56, Secs. 5.1.1.1-5.1.1.2]. Our implementation also provides 256-bit collision resistance regarding saved password hashes (see Table 14 in Appendix P) and can therefore be considered to have sufficient *Mis-Authentication Resistance* regarding the explicit sign-up/sign-in authentication. The only exception to this is the lack of a failed authentication limit, which must be imposed in a real-world implementation. Regarding such an implementation, we also recommend providing the option to use two-factor authentication, which would also further reduce the risk of online attacks that are based on observing credential hashes (see Section 6.3.5).

[80] NIST also requires that compromised passwords be changed but this is outside the scope of this thesis.

Table 9: Compliance of password-based authentication with NIST recommendations

#	Measure	Description	NIST Requirement	Implementation	Applicable	NIST compliant	Rationale for being compliant / not compliant
1	Minimum length	Passwords are required to have a minimum length.	**Yes** (at least 8 characters)	**Yes** (but irrelevant)	**No**	**Yes** (but irrelevant)	The mimimum and maximum length, as well as the set of allowed characters do not have an impact on the password security, which only results from the number of randomly chosen password phrase words (nouns) and the number of words from which these are chosen (size of word database). However, the customer could be given the opportunity to increase the number of words in the passphrase if he wishes.
2	Maximum length	Passwords should be allowed up to a certain a minimum length.	**No** (but 64 characters are recommended)	**No** (but irrelevant)	**No**	**Yes** (not applicable)	
3	Allowed characters	All printing ASCII & unicode characters as well as space should be allowed to be used in the password.	**No** (but recommended)	No (but irrelevant)	**No**	**Yes** (not applicable)	
4	Minimum complexity	Passwords need to be sufficiently complex.	**Yes** (but without complexity rules)	**Yes**	**Yes**	**Yes**	Sufficient password complexity is achieved by randomly choosing 4 words from a large set of unique words.
5	No complexity rules	Complexity rules other than password length should not be imposed.	**No** (but recommended)	**No**	**No**	**Yes** (not applicable)	Customers cannot choose their password freely, but sufficient entropy is guaranteed nonetheless.
6	Password generation limit	The allowed number of password generations should be limited (not considered by NIST, since not applicable).	**No** (not applicable)	**No**	**Yes**	**No** (not applicable, but should be changed)	If customer can generate an unlimited number of passwords, there is a small chance that they more often would use passwords with certain nouns in it, respectively not choose passwords with certain other nouns in it. A resonable limit for example would be 100 (password entropy would be reduced only by factor 100).
7	No password hint	The customer may not be given the opportunity to specify a password hint that is accessible to anyone.	**Yes**	**Yes**	**Yes**	**Yes**	The NIST requirement is fufilled.
8	No specific information required	The sign-up / sign-in process may not prompt the customer to specify cer-tain information as his password (i.e. security questions are not allowed).	**Yes**	**Yes**	**Yes**	**Yes**	The NIST requirement is fufilled.
9	Password blacklist check	A chosen password should be checked against a blacklist of likely chosen passwords, "bad passwords" and previously breached password corpuses.	**Yes**	**No** (but irrelevant)	**No**	**Yes**	The customer cannot choose a bad password, since it is being generated for him. If the password was contained in a breached corpus, this would be pure chance.
10	Password selection guidance	The customer should be supported in using a strong password, e.g. by displaying a password-strength meter.	**No** (but recommended)	**Yes**	**Yes**	**Yes**	The password generator ensures that the user selects a strong password. Therefore the NIST requirement is fufilled.
11	Rate limiting	The customer should be limited to a certain number of failed authentication attempts (100 as a maximum).	**Yes**	**No**	**Yes**	**No**	A real-world implementation of the platform should implement a rate limiting mechanism.
12	Allow pasting passwords	Customers should be able to paste their password, e.g. from a password safe application.	**No** (but recommended)	**Yes**	**Yes**	**Yes**	The NIST requirement is fufilled.
13	Unmasking option	Customers should be able to reveal the characters / password they entered (instead of dots or asterisks) in order to check their input.	**No** (but recommended)	**Yes**	**Yes**	**Yes**	The NIST requirement is fufilled. A custom JavaScript & CSS based function was implemented for this.
14	Authenticated encryption	The password should be trasmitted over an encrypted, authenticated channel.	**Yes**	**Yes**	**Yes**	**Yes**	The NIST requirement is fufilled (https with TLS 1.2 and secure ciphersuite).
15	Salted password hashes	Passwords should be saved as salted hashes using a suitable password derivation function in order to mitigte the risk of offline attacks.	**Yes**	**Yes**	**Yes**	**Yes**	The NIST requirement is fufilled. On the client side we perform 10.000 iterations of PBKDF2 with a random salt of 32 bytes (stored encrypted).
16	Secret salted password hashes	An additional round of key derivation functions should be performed with a secret salt.	**No** (but recommended)	**Yes**	**Yes**	**Yes**	The NIST requirement is fufilled. On the server side we perform 100.000 iterations of PBKDF2 with a random salt of 32 bytes (stored encrypted). NIST requires that the salt is saved separately from the hash so that it cannot be easily recovered, which we do not. However, both, salt and the hash itself are stored encrypted, which we argue also fulfills the requirement.
17	Side-channel attack resistance	Passwords may not be recovered by an attacker performing a side-channel attack, e.g. by analyzing the amount of time it takes the server to reject a certain password (hash).	**No** (but recommended in different context)	**Yes** (in parts)	**Yes**	**Yes** (not applicable)	NIST does require this measure only in certain contexts. However, we used a constant time comparison for checking password hashes.

In addition to the explicit authentication through the sign-up and sign-in process, we must also examine the implicit authentication which is given through the usage of web sessions. That is that no one should be able to pose as if they were the legit owners of a certain web session which in fact does not belong to them. For common websites, ensuring this is especially important once the respective customer authenticated via sign-in or sign-up, since an attacker would then gain access to particularly sensitive information (e.g. account information) and could possibly perform actions that they are not authorized for (e.g. ordering products). In contrast, for the *GhostBuy Platform* it is almost equally important to ensure that nobody gains access to a web session before the customer authenticated themselves, since the customer's product searches – which they can perform without explicit authentication – are considered to be very sensitive information as well.

In Table 10 we summarized the measures that are (to be) implemented in the *GhostBuy Prototype* to prevent certain attacks that target the implicit authentication through web sessions. In particular, we have made guessing and predicting session IDs very difficult by increasing the session ID size to match the key size of the AES256 key and configuring that a *Cryptographically Secure Pseudorandom Number Generator (CSPRNG)* is used for their generation (see Table 14 in Appendix P).

We argue that guessing/predicting a session ID therefore should be as difficult as it would be to guess or "predict" an AES256 key. To mitigate the risk of a session fixation attack, we suppressed the default functionality of the web session manager that allows session IDs to be specified in the URL parameters. We also implemented several effective measures against *Cross Site Scripting (XSS),* but additional measures could be implemented as described in Table 10.

Table 10: Overview of (to be) implemented measures against mis-authentication attacks

Attack	Description	(To be) implemented counter measure(s)	Implemented (where)
Session ID guessing / prediction	An attacker guesses / predicts the session ID of a customer and establishes a connection with that session ID.	The session ID length was increased from 128 bits to 256 bits in order to match the key size of the AES256 encryption.	**Yes** (*Portal Application* ->Tomcat Server Configuration)
		The web server has been configured to use a cryptographically secure random number generator ("Windows-PRNG") so that predictions, based on session IDs that were assigned before cannot be made.	**Yes** (*Portal Application* ->Tomcat Server Configuration)
Session fixation	An attacker initializes a web session and tricks a customer into using that session, e.g. by sending them a link which has the Session ID set via URL-parameters.	The platform does not allow for using session IDs in the URL-parameters. IDs are only set via cookies and sessions in which http requests try to append session IDs as URL-parameters are directly invalidated.	**Yes** (*Portal Application* ->Session Filter)
Cross Site Scripting (XSS)	An attacker forges a URL with certain parameters that would be displayed on the target page but actually contains for example JavaScript code (non-persistent). Alternatively the attacker can store the code in the website, e.g. by submitting it as a comment in a user forum. Hereby, the attacker for example could access the content of the session cookie.	The website does not display (reflect) any URL-parameters. The website does not store/show customer input in areas that are visible to other customers.	**Yes** (*Portal Application*)
		The SameSite cookie parameters has been set to "strict". This avoids that malicious code executed on other websites can access the cookie's content.	**Yes** **(***Portal Application* -> Configuration of context.xml in META-INF)
		Configured the built-in TomCat HttpHeader-SecurityFilter to prevent certain XSS attacks	**Yes** **(***Portal Application* -> Configuration of web.xml in WEB-INF)
		Previous search terms are displayed by dynamically loading them into an input field (i.e. code is not executed).	**Yes** **(***Portal Application*)
		Implement Content Security Policy (CSP) http headers to block cross site scripting	**No**
		The *GhostBuy Platform* should perform server side escaping / sanitization of user inputs.	**No** (implemented only client side input checks for fields other than search box, rudimentary checks in *Backend Application*)
All above	See attack descriptions above	Session IDs are changed with each page request.	**Yes** (*Portal Application* ->Session Filter)
		The Initial Cryptographic Setup ensures that the impact of any successful misauthentication attack described above would be limited, since the attacker does not obtain knowledge about (the product-related) information exchanged with the *Backend Application* . For this, the attacker would need to obtain the *Backend-Session Key* (6-B/C).	**Yes** *Portal Application* , *Backend Application*

Finally, we implemented two measures that are effective against all the above attacks:

- Firstly, we implemented that session IDs are changed with each page request. This is generally a common method but to the best of our knowledge it is usually performed only once, when a user signs up or in. Our reasoning for doing this for every page request is that even if an attacker was able to determine a customer's session ID once, they cannot use it for long: The next page request – and thus the next session ID change – will effectively either block the attacker from the customer's session (when the customer performs a page request) or block the customer from their own session (when the attacker performs it).
- Secondly, even if an attacker was able, for example, to repeatedly predict a session ID, the *Initial Cryptographic Setup* itself would at least protect all product-related information from being obtained by the attacker and also would retain them from performing any actions on the customer's behalf: This is, since all product-related information is encrypted with the *Backend Session Key (6-B/C)* which cannot be easily retrieved by the attacker unless they have (remote) access to the customer's device or to another application of the *GhostBuy Platform* (see step 7 in Section 5.5 and also Figure 37).

In conclusion, we do not claim that these measures (see Table 10) protect against all possible attacks targeting the implicit authentication through web sessions but that they at least make it significantly harder to carry out a successful attack. 256-bit security is provided against the attacks that we considered, respectively against attacks that do not bypass the above measures.

In summary, we note that the *GhostBuy Prototype* provides strong *Mis-Authentication Resistance*, regarding both explicit and implicit authentication, although further improvements could be made, especially for a real-world implementation of the platform.

6.4. Deployability

The final criterion against which the *GhostBuy Platform Architecture* and *Prototype* shall be evaluated is *Deployability*. As the name suggests the criterion asks whether *GhostBuy* could be implemented and operated as a real-world platform. For the evaluation we examine if it provides *Basic Functionality and Convenience*, if it has a good *Computational Efficiency* and

Cost Efficiency and if it – from the perspective of the companies it interacts with – does realize *Integration Transparency.* Finally, we also evaluate if and why the threat model described in Section 6.1 is a *Real-World Threat Model.*

6.4.1. Basic Functionality

The property *Basic Functionality* states:

> *The platform must be able to provide the Basic Functionality of an online web shop (browsing, ordering, payment, delivery and returns) – but in an anonymous fashion.*

We have shown that the *GhostBuy Prototype* implements the functionalities browsing, ordering and payment of products with the sole limitation being that the payment and ordering is simulated in the respective web services by "checking" the credit card information and writing the order information to a log file – instead of for example incorporating the web service APIs of a payment processor such as PayPal and an e-commerce marketplace such as eBay. The reason for this is that such an implementation would have required to be assessed by the respective company if one is compliant with their requirements, which was obviously not possible to do within the scope of this thesis. We did, however, successfully trigger orders in the eBay sandbox environment by using their so-called *API Explorer*. Nevertheless, we did not implement this functionality in the prototype since it does not yield much more than a generic web service response. In addition, we already showed that the platform in general can use the eBay API by performing the *GhostBuy Prototype's* product searches in the eBay production environment.

We described how deliveries would be handled by the *GhostBuy Platform* with the most critical steps being the platform-internal package handover (see Section 5.7) and the customer's verification of package receipt (see Section 5.10). To handle returns, the process – i.e. in particular the platform-internal package handover – would be inversed.

The above shows that the *GhostBuy Architecture* can very well provide the Basic Functionality of an online shop – even though the *GhostBuy Prototype* does not implement all of it – and in Section 6.2 (including subsections) we showed that this is done in an anonymous/privacy-preserving fashion.

6.4.2. Convenience

To fulfill the property *Convenience* the following must be given:

> *The platform must provide the Basic Functionality of an online shop without requiring the customer to use additional tools to do so.*

This means that the whole purchase process is handled by the platform itself, i.e. that it can/must trigger and organize external processes such as the delivery of the purchased products. The underlying intention hereby is that a customer has basically the same experience as if he was shopping non-anonymously at a common online retailer such as amazon.ca or bestbuy.ca.

Throughout this thesis we have shown that the *GhostBuy Platform* is the central instance for all parties involved in the purchase process, keeping in mind that due to the underlying concept of *Separation of Data* the *Frontend Company* and *Backend Company* interact with separate parties. From the *GhostBuy Platform* customers' point of view, the fact that they can not only complete all purchase steps via one platform, but also do not have to use specialized tools such as the Tor browser for this purpose, is a positive factor. In fact, to use the platform`s *Client Application* / *Portal Application*, they can choose between many popular browsers. We therefore note that the customers' user experience is similar to using a common online shop – though admittedly *GhostBuy* has certain platform-specific features.

6.4.3. Computational Efficiency

We defined the property *Computational Efficiency* to be fulfilled if the following is true:

> *The platform prototype must be able to run on a standard home office PC and complete the respective transactions – i.e. each step of a purchase process – within a period of time that is acceptable to consumers, e.g. less than 10 seconds.*

However, we argue that to properly evaluate the *Computational Efficiency* we also need to consider the amount of data transferred between the *Client Application* (customer's browser) and the *Portal Application* and the amount of local storage consumed by the *Client*

Application. Finally, we also briefly analyzed how the total execution time of the purchase process steps is composed.

We measured the performance of the *GhostBuy Prototype* by performing 16 (+1) consecutive browsing flow steps of a purchase process, with the platform applications (Portal/Frontend/Backend, including SQL servers) and the *Client Application* (browser) running on the same system. The browser's cache was cleared before the test. The results of the measurements can be seen in Table 11.

We would like to point out the following five key-takeaways:

1. The total time consumption – from the customer perspective – of each step to be performed on the *GhostBuy Platform* is lower than 5 seconds, in most cases even lower than 3 seconds (numbers highlighted in bold font). It should be noted that the customer will see the platform reacting to their actions much faster than this, since either the target page will open directly (and subsequently will load the encrypted content) or the respectively clicked button will directly show an appropriate label like e.g. "Signing in…" until the target page is opened.
2. In about half of the steps the *Backend Application's* request processing accounts for most of the time consumption, with the largest part resulting from querying the online shop (eBay) and the subsequent downloading (and scaling – but this is fast) of the product images. Please consider the *Client Application's* "Idle Time" and the respective processing times of the *Backend Application* left and right of the bold black vertical line to see this correlation. It can also be seen that the *Backend Application's* processing time is much lower when it does not incorporate querying the online shop and the processing of images. This for example is the case for product quantity modifications and the checkout process.[81] Therefore the overall time consumption for product quantity modifications in the shopping cart is even lower as compared to adding the very same product again, but the number of bytes transferred to the *Client Application* stays about the same (highlighted in blue / marked with letter A).

[81] A real-world implementation might double-check product prices and availabilities at that time.

Table 11: Performance measurements of GhostBuy Prototype on desktop PC

#	Browsing Flow Step	Description	Run	Products retrieved in this step			*Client Application* (Browser)													*Backend Application*			
							Resources			Time Consumption in ms				Scripting						Approximate Time Consumption in			
				Retrieved #	Image size after scaling	Information & image source	Bytes loaded	Bytes transferred	Bytes / # of images	Total	Time / # of Images	System (e.g. profiling)	Loading (e.g. parse html, CSS)	Microtasks (execution)	Other (e.g. compile)	Rendering	Painting	Key Generation + Decryption (if appl.)	Idle (waiting)	Total	Query online shop (eBay)	Image retrieval & scaling	Other (e.g. logging, processing, encryption)
1	Initial Cryptographic Setup	Open portal URL (➔ key generation & key exchange, shopping cart creation, home page load)	1st	n/a	n/a	n/a	D 6,424,731	2,083,375	n/a	**3,364**	n/a	280	103	113	304	155	13	2,070 B + 22	304	5	n/a	n/a	5
2	Home Page Reload	Click logo on home page (➔ home page reload)	(1st)	n/a	n/a	n/a	2,428,812	15,791	n/a	**444**	n/a	62	26	110	121	18	4	n/a	103	processed by *Portal Application*			
3	Product Search	Search for "EBT" in shop "eBay Canada"	1st	E 20	small	eBay	3,072,111	E 652,926	32,646	**1,614**	81	121	45	234	195	59	17	n/a	943	839	501	328	10
4	Home Page Load & Shop Preselection	Click logo link in shop page header (➔ home page load, preselects shop)	(1st)	n/a	n/a	n/a	2,348,484	15,649	n/a	**455**	n/a	74	27	111	124	15	4	n/a	100	processed by Portal Application			
5	Product Search	Search for "EBT" in shop "eBay Canada"	2nd of #3	E 20	small	eBay	2,990,225	E 389,371	19,469	**1,681**	84	136	45	267	192	63	20	n/a	958	867	550	312	5
6	Product Search	Search for "EBT" in shop "eBay Canada" by selecting "100 products/page"	(1st)	100	small	eBay	4,387,579	1,900,148	19,001	**4,234**	42	197	81	588	183	94	35	n/a	3,056	1966	540	1307	119
7	Product Search	Search again for "EBT" in shop "eBay Canada" by retriggering search in shop page header	Similar to #6	100	small	eBay	4,795,885	1,893,604	18,936	**3,075**	31	184	124	580	76	131	41	n/a	1,939	1536	601	825	110
8	Product Details	Open product details of "HOn30 East Broad Top 23 1/2 Foot Early Flatcar Kit 2 Pack EBT"	1st	10	large	eBay	3,502,124	894,527	F 89,453	**2,432**	243	138	40	199	184	57	13	n/a	1,801	1535	1535		
9	Add to Shopping Cart	Add product "HOn30 East..." to shopping cart	1st	1	med.	eBay	2,637,517	184,401	184,401	**1,724**	1,724	113	33	126	177	27	13	n/a	1,235	1078	1070		8
10	Product Details	Open product details of "HOn30 East ..." (by clicking image in shopping cart)	2nd of #8	10	large	eBay	3,501,610	789,314	F 78,931	**2,464**	246	118	38	162	173	57	15	n/a	1,901	1636	1636		
11	Add to Shopping Cart	Add product "HOn30 East..." to shopping cart	2nd of #9	1	med.	A eBay	2,637,541	A 107,695	107,695	A **1,626**	1,626	105	31	121	152	48	15	n/a	1,154	A 974	972		2
12	Decrease Item Quantity	Decrease item quantity of product "HOn30 East..." in shopping cart (1x = down to 1)	1st	1	med.	cart	2,637,444	107,538	107,538	**625**	625	94	34	126	136	52	8	n/a	175	2	n/a	n/a	2
12-2	Increase Item Quantity	Increase item quantity of product "HOn30 East..." in shopping cart (1x = up to 2)	(1st)	1	med.	cart	2,637,378	107,472	107,472	**680**	680	96	31	128	145	45	10	n/a	225	1	n/a	n/a	1
12-3	Decrease Item Quantity	Decrease item quantity of product "HOn30 East..." in shopping cart (1x = down to 1)	2nd of #12	1	med.	cart	2,637,385	107,480	107,480	**597**	597	89	29	123	137	43	9	n/a	167	0	n/a	n/a	0
13	Start checkout	Click "Checkout" (redirects to sign-in page)	(1st)	n/a	n/a	n/a	2,334,756	60,867	n/a	**366**	n/a	74	27	58	106	3	5	n/a	93	processed by Portal Application			
14	Sign-In	Perform Sign-In (click "sign in" after entering test account data)	(1st)	1	med.	cart	2,461,898	132,943	132,943	**2,365**	2,365	102	29	91	156	62	14	C 91	1,820	processed by *Portal & Frontend Application*			
15	Continue Checkout	Click "Continue Checkout" (after entering shipping and payment information)	(1st)	1	med.	cart	2,451,614	90,360	90,360	**1,222**	1,222	84	27	35	139	41	13	n/a	883	1	n/a	n/a	1
16	Place Order	Click "Place your order"	(1st)	n/a	n/a	n/a	1,746,152	8,616	n/a	**932**	n/a	58	22	26	42	34	8	n/a	742	*no exact time stamps in log*			
17	Initial Cryptographic Setup	Open portal URL (➔ key generation & key exchange, shopping cart creation, home page load)	2nd of #1	n/a	n/a	n/a	D 6,464,119	35,597	n/a	**2,169**	n/a	190	80	115	246	66	9	1238 B + 22	203	11	n/a	n/a	11

3. A notable exception to this is the execution of the *Initial Cryptographic Setup* during which the RSA key generation performed by the *Client Application* is the longest running task (highlighted in yellow / marked with letter B). It should be noted that *Web Crypto API* processing times without further analysis are counted as idle times by the Chrome Developer Tools that we used for our *Client Application*-related measurements. However, since most tasks performed by the *Web Crypto API* – other than the RSA key generation and decryption – take less than 1 millisecond to complete, we did not visualize these times separately in Table 11. An exception to this is the time consumption of the generation of the *Client-side Secret Key (5-C)* (highlighted in orange / marked with letter C). Considering that this process (100,000 iterations of the PBKDF2 algorithm) only took 91 milliseconds to complete, we note that we could have chosen more demanding parameters (e.g. 1 million iterations), going beyond the recommendations of NIST [56, Secs. 5.1.1.1-5.1.1.2].

4. The first run of each browsing flow step includes downloading the respectively required web page resources – such as HTML/CSS/JavaScript files and logos – during the target page load which, however, does not very much increase time consumption but rather the amount of *Transferred Bytes* – as opposed to the *Bytes loaded* (decompressed size & also resources loaded from cache). This can be seen in particular from the first and second run of the *Initial Cryptographic Setup* (highlighted in violet / marked with letter D). Nevertheless, the second run of a certain step in two cases was completed significantly faster. Although our limited measurements do not give a clear reason for this, it seems that only little, if any, performance gains result from unencrypted resources loaded from the browser cache during the second run. Larger gains may have resulted from *Web Crypto API* libraries being already loaded on the first run (consider the RSA key generations in step 1 and 17). In the case of the performance gain from step 6 to 7 the fact that – likely by chance – images were downloaded faster during the second run had a much bigger impact. Additionally, a more detailed analysis of these steps showed that the remaining majority of the performance gain actually resulted from the steps not being exactly the same: The selection of 100 products/page in step 6 resulted in about 700 ms delay (“Idle”) as compared to the retriggering of the search in step 7.

5. It can also be seen that most transferred bytes – especially after required web page resources were initially loaded during a first run – result from the encrypted product images (and product descriptions). The first and second runs of searches with 20, respectively 100 product results (highlighted in darker green / marked with letter E) show that the number of transferred bytes scales with the number of retrieved products, after the required web page resources have been loaded during the first run with 20 results. It should be noted that product images naturally disproportionately contribute to the number of bytes transferred as can be seen from the increased amount of data transferred associated with larger product images on the *Product Page* (highlighted in brighter green / marked with letter F).

The development system that we used for above measurements shown in Table 11 – a desktop PC featuring a 6-core Intel Xeon CPU running at 3.47 GHz and 12 GB of DDR3 memory running at 1,600 MHz, effective (PC3-12800) – cannot be considered to be a standard home office PC, which however we wanted to utilize for evaluating the *Computational Efficiency*. Therefore, we repeated the measurement on a laptop that featured a 2-core Intel Core i7 7500U mobile CPU running at 2.7 GHz (up to 3.5 GHz) and 8 GB of DDR4 memory running at 2,133 MHz, effective (PC4-19200). For comparison, the desktop CPU is about 95 % higher rated in overall benchmarks as compared to the laptop CPU, but about 20 % lower rated for single thread applications.[82]

The results of this verification measurement (see Table 15 in Appendix R) show that the 5 longest running browsing flow steps complete also well below 10 seconds on the slower laptop system, though they take about 0.8 to 2.1 seconds longer (i.e. 25% to 60% longer) than on the desktop system. Interestingly, the RSA key generation that is performed by the *Client Application* took 50% to 65% longer, but the RSA decryption took about 50% less time. Also, the PBKDF2 key derivation that is performed to generate the *Client-side Secret Key (5-C)* takes 20% less time. The exact reasons for this behaviour could only be found with further analysis, but we assume that the in general slower performance of the laptop environment is

[82] See https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+W3690+%40+3.47GHz&id=1275 and https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i7-7500U+%40+2.70GHz&id=2863

related to the fact that the desktop system has more CPU cores which can therefore distribute the workload of the running server applications (Portal/Frontend/Backend, including SQL servers) and the *Client Application* to more cores / more evenly, especially when these are either parallelized (such as the image processing) or long running (such as the RSA key generation). In contrast, the faster execution of short running, non-parallelized or memory intensive tasks such as RSA and AES encryptions and decryptions as well as the PBKDF2 key derivation might result from the more modern CPU architecture and the higher memory frequency of the laptop system.

Finally, we want to point out that the *Client Application* also works efficiently regarding the browser's usage of local storage: Although each unique parameter configuration of a web page opened by the *Client Application* requires an associated, encrypted parameter set to be saved in the *IndexedDB*, this does not result in a lot of storage consumption. For example, we found that opening 114 different pages only results in an *IndexedDB* size of about 159 kB (see Figure 41 in Appendix Q). The *IndexedDB* is cleared when the *Client Application* rebuilds the *Initial Cryptographic Setup*. The data recorded for our measurements can be found in Appendix W.

6.4.4. Cost Efficiency

To ensure that the platform could be operated in the real-world we must evaluate its *Cost Efficiency*:

> *We consider the property Cost Efficiency to be fulfilled if the platform could be reasonably financed for example by fees or provisions.*

We want to add that this property is rather subjective and that we do not consider it to be as important as the other properties.

While it would be out of the scope of this thesis to perform a thorough *Break-Even Point (BEP)* calculation, we want to list certain qualitative key factors that would have to be taken into account for such an analysis:

- Since packages are shipped twice during the *GhostBuy Platform's* purchase process – once from the online shop to the *Backend Company* and once from the *Frontend Company* to the customer – shipping costs could theoretically double, but certain

economies of scale should reduce that rate: For example, some online shops offer free shipping for frequent customers or for "premium customers". As the first package recipient is always the *Backend Company*, these scale effects should apply for the first shipment.

- Other scale effects could potentially lower the product costs themselves, or create additional revenue for the platform, such as discounts or payback programs for frequent customers.
- These economies of scale and additional revenue sources cannot be factored in if the respective online shop has policies in place that would render the *Backend Company* ineligible for these.
- For the second shipping, rates could be bargained with the respective shipper(s).
- Purchasing digital products on the platform would not create real additional shipping costs (unlike physical products that would be physically shipped twice). Please see also Section 6.5.2.
- Operating the *GhostBuy Platform* should not be cost intensive, otherwise. The previous section showed that the platform could presumably be run on inexpensive hardware. The catering to a larger number of customers would require better hardware but would also generate more income. Regarding labour costs we assume that the company could theoretically be run by two people in the beginning (one for the *Frontend Company* and one for the *Backend Company*) but that it is more likely to be run by at least 2 to 3 persons per company (e.g. 1 x IT, package handling & monitoring, 1 x customer service, accounting & monitoring), especially when the number of customers increases.

Nevertheless, we assume that it is likely that *GhostBuy* customers would have to pay a fee to cover the additional shipping, labour, and operational costs. It would have to be examined how high these costs would be and whether customers would be willing to pay this additional amount for preserving their privacy and anonymity.

6.4.5. Integration Transparency

The property Integration Transparency is fulfilled if interacting with the Frontend Company and Backend Company does not require any technical changes at the involved platform of external parties (i.e. online shops, shippers and banks).

We have shown that the *GhostBuy Platform* shall use existing web services and other APIs to connect to online shops, shippers and banks (e.g. see Sections 3.1 and 5.1) and it is common knowledge that banks and shippers provide APIs especially for their business customers (such as *GhostBuy*). Furthermore, the *GhostBuy Prototype* implementation demonstrates the utilization of an e-commerce platform API (eBay). Other online shops, such as Amazon and Walmart do also provide APIs for browsing their catalogues, but not for ordering goods. Nevertheless, it would be possible to order from them by using third-party APIs such as the Zinc API[83] which would however increase the operational costs of the *GhostBuy Platform* and introduce another party into the process (to be considered regarding privacy).

6.4.6. Real World Threat Model

The final property to consider in terms of the *GhostBuy Platform's Deployability* is whether the threat model that we described in Section 6.1 is actually a *Real World Threat Model*. We argue that this is the case for the following reasons:

- The parties involved in the purchase process are all honest but curious. This corresponds to reality in that if they were dishonest, they would risk being sanctioned, for example, by being excluded from future purchases (party) or being dismissed (employee). On the other hand, it is also realistic that we have described them as curious, because for these companies, collecting information is important to better understand their customers and thus improve their services and make more profit – for example by selling customer information.

[83] See zincapi.com

- The described external adversaries are neither unrealistically weak nor unrealistically strong: For example, they target the least secure – though not insecure – parts of the platform and carry out attacks that have realistic but distinct goals (e.g. obtaining information about a certain customer vs. obtaining information about all customers). However, like the other parties involved, the adversaries are computationally bounded and therefore unable to break state of the art encryption algorithms such as AES256.

6.5. Limitations & Possible Improvements

Although we have described certain drawbacks and limitations of the *GhostBuy Architecture*, respectively the *GhostBuy Prototype* in earlier sections, we would like to summarize these here but also discuss further important limitations. We cover limitations that are inherent to the proposed architecture, and limitations that are specific to our prototype implementation in two separate sections.

In this context, we would like to emphasize once more that the specific *GhostBuy Architecture* and *the GhostBuy Prototype* represent only one of probably many possible implementations that realize our proposed design, as described in Section 3.1 and shown in the architectural blueprint in Figure 5. This means that the *GhostBuy Architecture* and *GhostBuy Prototype* as we describe it in Chapter 5 could very well be modified by using different technologies for certain or even all applications that are shown in Figure 8. E.g., the database engine MySQL could be replaced by MongoDB and an alternative ORM – or no ORM at all – could be used. Moreover, the architecture could presumably even be realized with a completely different design pattern instead of our proposed extended MVC pattern. In this case, however, we assume that the encryption sequences and data flows would be different from those described in Sections 5.4 to 5.6.

6.5.1. GhostBuy Architecture Limitations

An overall limitation of the *GhostBuy Architecture* is that it can only be implemented in real life if the parties involved – especially the online shops – have no objections and legal arguments against this, since for example they might disapprove of not obtaining information about their customers and therefore object against *GhostBuy* using their services. In addition – even if third parties allow for using their services – it is possible that functional limitations

of their APIs would effectively limit *GhostBuy,* too: During the implementation we experienced for example that the eBay API did not consistently sort and return search results regarding prices especially regarding included shipping costs. This issue might have been related to the then-Beta-status of the API and could mostly be overcome by postprocessing the results in our implementation. A real-world implementation, however, should use reliable APIs to yield reliable results (see 6.3.1).

Another limitation or rather drawback of the architecture is that all processes are at least doubled as compared to common online shopping and the *GhostBuy Platform* has to perform its own purchase processing as well. Therefore, orders performed on *GhostBuy* will take longer until they are delivered and generate more environmental pollution due to being shipped twice, the costs will potentially be higher, the repackaging will increase waste amounts and running the *GhostBuy Platform* will increase energy consumption and therefore environmental pollution, too. It should be noted, though, that these are inevitable "costs" for using a service like *GhostBuy* but that their extent could partly be reduced by appropriate measures such as using economies of scale (see Section 6.4.4), recycled packaging and electricity from renewable sources.

Finally, the *GhostBuy Architecture* also has an inherent privacy & anonymity limitation: The parties involved could collaborate to find correlations in their data: For example, Bank 1 and an online shop could try to link a customer's payment to *GhostBuy* (via Bank 1) to the respective order at the online shop. However, this kind of collaboration would involve a lot of effort and counter measures can be taken (see Section 6.2.1).

6.5.2. GhostBuy Prototype Limitations

Although we have tried to implement real functionality in as many parts of the *GhostBuy Prototype* as possible, we have not been able to do so with regard to the execution of payment transactions and the triggering of orders (see Section 6.4.1). We also did not implement a functionality that would allow for the platform to decide whether customers are eligible to order certain goods via *GhostBuy* – e.g. regarding if they are of legal age.

Regarding our usage of encryption keys, our prototype has several limitations, respectively could be improved. First, we use certain keys directly to encrypt objects that are saved in the

databases (see Figure 37 in Section 6.3.5) – here, it would be better to use superordinate keys to encrypt those keys, respectively to implement a key management system/database, so that changing keys could be handled more efficiently. Second, we load pre-generated keys from Java keystores (see Table 4 in Section 5.4) for which the respective keystore-passwords and key-passwords are hard-coded into the source code. This should obviously be avoided for example by using a trusted device that provides keys, by using a key management system/database and/or by entering passwords manually, respectively by loading these from smartcards. Third, our implementation does not limit the duration or otherwise protect keys in memory – except for the *Client Wrapping Key (2-P/C)* that is used in the *Client Application* (see Section 5.9). A real-world implementation should handle this differently.

We also noticed that the lengths of plaintexts (= unencrypted data) directly correlate with the lengths of the ciphertexts (= encrypted data) when encrypted with the AES256-GCM algorithm. This is due to the fact that AES256-GCM internally works like a stream cipher and therefore does not pad ciphertexts to e.g. match AES-blocksizes. This circumstance would allow for drawing conclusions about certain plaintexts solely based on ciphertext lengths, especially when possible plaintexts are known and differ in size. Therefore, a custom padding should be implemented in the respective crypto classes (Java) and modules (JavaScript), because ensuring that known plaintexts (e.g. the selected shop) have the same length would otherwise be complex, error-prone or simply impossible.

Two further limitations result from the customer using a web browser to run the *Client Application.* First, the autofill function of web browsers can leak sensitive information such as search terms and shipping addresses to subsequent users of the same device and browser. Although this will in many cases be prevented by platform-external means – such as user-specific or reset browser profiles on public PCs – a real-live *GhostBuy Platform* implementation should seek to implement its own countermeasures: For example, it would be possible to create randomized, generic field identifiers so that browsers are unable to recognize the form field when the page is revisited later. However, actually avoiding that form field contents are saved at all is a better solution and could be achieved by the methods described in [63] or by not using html form submission methods in the implementation. Currently, we already intercept form submission methods for other reasons, but the form field contents are

still stored by browsers by default. A potential second browser related limitation is that decrypted content which has been populated on the web page somehow might be cached by the browser and therefore could be retrieved by an offline attacker. Based on our observations we doubt that this is the case, but it should be examined further.

Also, the encrypted databases that are used by the *Frontend Application* and *Backend Application* are potentially not very performant regarding heavier workloads, since e.g. database indexes are not being created and certain references are encrypted themselves, too. It could be examined whether the databases are nevertheless efficient/performant enough to handle normal workloads. Heavy workloads are not to be expected due to the lack of big data operations and a limited number of customers.

The *GhostBuy Prototype* does not support the delivery of digital products but this could be achieved for example by having the *Backend Application* encrypting electronic products with the *Backend Session Key (6-B/C)* and then delivering the goods to the customer via the *Portal Application*. It should be noted that in the case of delivering electronic products, certain disadvantages and limitations such as increased shipping costs and times would not apply or would apply only to a significantly lesser extent.

Finally, we would like to mention that a real-world implementation of a *GhostBuy*-like platform would have to convince its customers that it properly handles their personal information and e.g. actually does keep it separate in order to preserve their privacy (*Frontend Company*) and anonymity (*Backend Company*). Although being transparent about the platform's internal workings would be one option to give its customers this confidence, a more advanced, convincing and – once properly implemented and widely used – more convenient way to achieve this goal would be to employ privacy enforcement technologies such as those described in [64], [65].

We consider the above limitations as the most significant or least obvious limitations of the *GhostBuy Prototype Implementation*. We have described less significant or more obvious (prototypic nature of the implementation) limitations of the prototype – some of those having been discussed earlier – in Table 16 in Appendix S. The table also contains information regarding possible measures to overcome the respective limitations.

7. Conclusion & Future Work

In this thesis we have shown that consumers have a demand for anonymous online shopping but that the designs proposed in other work were unlikely to be implemented in the real world for various reasons such as a high integration effort – also for customers, non-compliance with existing laws or limited functionality. A revisit of the literature confirmed these findings.

We discussed that a real-world design can only provide a certain approximation to actual anonymity with regards to legal and practical reasons. We improved the design by Groppe et al. [2] to propose an *All-Steps Anonymous Purchase Platform* by integrating their concept of *Separation of Data* into a single platform – *GhostBuy* – which greatly reduces the integration effort for all parties involved. Within the platform, the *Separation of Data* ensures that product-related information stays unknown to the *Frontend Company*, hereby preserving privacy, and that the customer's identity stays unknown to the *Backend Company,* hereby preserving anonymity.

Our prototype implementation, the *GhostBuy Prototype*, shows the general feasibility and efficiency of the approach, also highlighting the wide-spread adoption and high performance of the *Web Crypto API*. Moreover, the prototype demonstrates that a platform like this can be realized in a user-friendly way which is similar to the look-and-feel of common online shops, but also offers its users additional support in staying safe on the platform – for example by optionally checking search terms for personal information and generating secure passwords. In combination, this can help to convince more consumers – who are, for example, not comfortable with tools such as the Tor browser – to use this platform to regain control over their personal data when shopping online. This is especially important in times where more consumers have no choice but to shop online, despite concerns about losing control of their private information if they do so at Big Data-enabled online retailers. For the prototype we also implemented a non-essential but working database support that uses application-level encryption and *Object-Relational Mapping*. We provided a rudimentary, but working implementation of the package-handover-interfaces and outlined the handover-workflow as well as the package verification performed by customers and the platform's handling of law enforcement inquiries.

We evaluated that our proposed *GhostBuy Architecture* and *Prototype* meets the previously

defined criteria *Privacy & Anonymity* – including the property of *Transaction Unlinkability -, Security* – including the resistance against cryptographic and mis-authentication attacks – and *Deployability* – including *Computational Efficiency.* Certain shortcomings – e.g. regarding the usage of RSA4096 and a minor leak of personal information – have been explained and possible ways to address these have been laid out. Other limitations were discussed, and possible improvements were outlined.

Since we were unable to do so in the scope of this thesis, future work could explicitly examine – for example by means of a survey – whether and to what extent people would be willing to pay extra fees for anonymity when shopping online, in order to verify and quantify the consumers' demand for anonymity.

Our prototype implementation could be improved for example by sanitizing the original product description in a way that better preserves images and layout. Also, at a more fundamental level, it can be explored whether (and how) an improved architecture could protect the customer's privacy even better, for example by achieving that the *Frontend Company* does not learn the amount paid by the customer – provided this would comply with legal requirements. When considering individual parts of the prototype implementation, we note that the generation of user-friendly-but-secure passwords might be researched and improved in future work. A detailing and (prototypic) implementation of the outlined interface for law enforcement authorities could potentially improve the protection of personal information and customer privacy in existing applications (no unrelated data leaking to the authorities and investigated records stay unknown to application owner). Since we are sure that better, full featured solutions for application-level database encryption already exist, we would only like to suggest to examine our proposed approach for its applicability with respect to Java database applications that are lightweight but still have a high need for data security, such as the patient database of a mobile medical team.

Finally, we want to emphasize that the *Web Crypto API*, which served our purposes very well and can potentially be used for other innovative applications, requires constant qualified development and reviewing, for example to support post quantum cryptographic algorithms or to address general shortcoming such as its dependency on the delivery and execution of non-tampered JavaScript files.

8. References

[1] "How Many Products Does Amazon Sell? – April 2019," *ScrapeHero*, 24-Apr-2019. [Online]. Available: https://www.scrapehero.com/number-of-products-on-amazon-april-2019/. [Accessed: 05-Dec-2019].

[2] S. Groppe, F. Kuhr, and M. A. Coskun, "Anonymous Shopping in the Internet by Separation of Data," *Open J. Web Technol.*, vol. 5, no. 1, pp. 14–22, 2018.

[3] F. Willems, "Literature Review," *EBC7101: Research Workshop in E-Business Technologies*. (see Appendix W), 2019.

[4] M. Rennhard, S. Rafaeli, L. Mathy, B. Plattner, and D. Hutchison, "Towards Pseudonymous e-Commerce," *Electron. Commer. Res.*, vol. 4, no. 1–2, pp. 83–111, 2004.

[5] F. Belanger, J. S. Hiller, and W. J. Smith, "Trustworthiness in electronic commerce: the role of privacy, security, and site attributes," *J. Strateg. Inf. Syst.*, vol. 11, no. 3, pp. 245–270, 2002.

[6] E. Androulaki, "A Privacy Preserving ECommerce Oriented Identity Management Architecture," ProQuest Dissertations Publishing, Ann Arbor, 2011.

[7] T. Li, R. Zhang, and Y. Zhang, "PriExpress: Privacy-preserving express delivery with fine-grained attribute-based access control," in *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016, pp. 333–341.

[8] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–78, Dec. 2007.

[9] J. Foertsch and M. A. Gernsbacher, "In Search of Gender Neutrality: Is Singular They a Cognitively Efficient Substitute for Generic He?," *Psychol. Sci.*, vol. 8, no. 2, pp. 106–111, Mar. 1997.

[10] A. Bodine, "Androcentrism in prescriptive grammar: singular 'they', sex-indefinite 'he', and 'he or she,'" *Lang. Soc.*, vol. 4, no. 2, pp. 129–146, 1975.

[11] B. M. Bjorkman, "Singular they and the syntactic representation of gender in English," *GLOSSA-A J. Gen. Linguist.*, vol. 2, no. 1, Sep. 2017.

[12] "Singular 'they,'" *American Psychological Association*, 2019. [Online]. Available: https://apastyle.apa.org/style-grammar-guidelines/grammar/singular-they. [Accessed: 23-Feb-2021].

[13] "Legistics - Singular 'They,'" *Government of Canada Department of Justice*, 2020. [Online]. Available: https://www.justice.gc.ca/eng/rp-pr/csj-sjc/legis-redact/legistics/p1p32.html. [Accessed: 23-Feb-2021].

[14] "They," *Merriam-Webster.com Dictionary*. [Online]. Available: https://www.merriam-webster.com/dictionary/they. [Accessed: 23-Feb-2021].

[15] "Gendered Pronouns & Singular 'They,'" *Purdue Writing Lab*. [Online]. Available: https://owl.purdue.edu/owl/general_writing/grammar/pronouns/gendered_pronouns_and_singular_they.html. [Accessed: 23-Feb-2021].

[16] S. Ha and L. Stoel, "Consumer e-shopping acceptance: Antecedents in a technology acceptance model," *J. Bus. Res.*, vol. 62, no. 5, pp. 565–571, 2009.

[17] I. P. Riquelme and S. Román, "Is the influence of privacy and security on online trust the same for all type of consumers?," *Electron. Mark.*, vol. 24, no. 2, pp. 135–149, Jun. 2014.

[18] M. A. Eastlick, S. L. Lotz, and P. Warrington, "Understanding online B-to-C relationships: An integrated model of privacy concerns, trust, and commitment," *J. Bus. Res.*, vol. 59, no. 8, pp. 877–886, 2006.

[19] G. Bansal, F. M. Zahedi, and D. Gefen, "Do context and personality matter? Trust and privacy concerns in disclosing private information online," *Inf. Manag.*, vol. 53, no. 1, pp. 1–21, 2016.

[20] M. S. Ackerman, L. F. Cranor, and J. Reagle, "Privacy in E-Commerce: Examining user scenarios and privacy preferences," in *Proceedings of the 1st ACM Conference on Electronic Commerce*, 1999, pp. 1–8.

[21] Y. B. Limbu, M. Wolf, and D. Lunsford, "Perceived ethics of online retailers and consumer behavioral intentions," *J. Res. Interact. Mark.*, vol. 6, no. 2, pp. 133–154, 2012.

[22] A. D. Miyazaki and A. Fernandez, "Consumer Perceptions of Privacy and Security Risks for Online Shopping," *J. Consum. Aff.*, vol. 35, no. 1, pp. 27–44, 2001.

[23] G. Bella, R. Giustolisi, and S. Riccobene, "Enforcing privacy in e-commerce by balancing anonymity and trust," *Comput. Secur.*, vol. 30, no. 8, pp. 705–718, 2011.

[24] E. B. Sasson *et al.*, "Zerocash: Decentralized Anonymous Payments from Bitcoin," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 459–474.

[25] T. Hataguchi, "Anonymous Purchase And Sale System For Online Shopping And Delivery Services Via Computer Networks," United States Patent No. US 2001/0029472 A1, 2001.

[26] H. Yun, G. Lee, and D. J. Kim, "A chronological review of empirical research on personal information privacy concerns: An analysis of contexts and research constructs," *Inf. Manag.*, vol. 56, no. 4, pp. 570–601, Jun. 2019.

[27] M. Kolotylo-Kulkarni, W. Xia, and G. Dhillon, "Information disclosure in e-commerce: A systematic review and agenda for future research," *J. Bus. Res.*, vol. 126, pp. 221–238, 2021.

[28] A. Rial, "Privacy-preserving e-commerce protocols," *KU Leuven*, 2013.

[29] R. Bandara, M. Fernando, and S. Akter, "Explicating the privacy paradox: A qualitative inquiry of online shopping consumers," *J. Retail. Consum. Serv.*, vol. 52, p. 101947, 2020.

[30] T. Regner and G. Riener, “Privacy Is Precious: On the Attempt to Lift Anonymity on the Internet to Increase Revenue,” *J. Econ. Manag. Strateg.*, vol. 26, no. 2, pp. 318–336, 2017.

[31] X. Chen, S. Fang, Y. Li, and H. Wang, “Does Identification Influence Continuous E-Commerce Consumption? The Mediating Role of Intrinsic Motivations,” *Sustain. (Basel, Switzerland)*, vol. 11, no. 7, p. 1944, 2019.

[32] A. Ben Ayed and M. A. Belhajji, “Trade Route: An Anonymous P2P Online Exchange Market Using Blockchain Technology,” in *BT - Integrated Science in Digital Age 2020*, 2021, pp. 3–9.

[33] Y. Jiang, C. Wang, Y. Wang, and L. Gao, “A Privacy-Preserving E-Commerce System Based on the Blockchain Technology,” in *2019 IEEE 2ND INTERNATIONAL WORKSHOP ON BLOCKCHAIN ORIENTED SOFTWARE ENGINEERING (IWBOSE)*, 2019, pp. 50–55.

[34] A. Mars and W. Adi, “Fair Exchange and Anonymous E-Commerce by Deploying Clone-Resistant Tokens,” in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 1226–1231.

[35] M. Ike and K. Sarac, “PPEP: A Deployable Privacy Preserving E-Commerce Protocol for Electronic Goods,” in *Proceedings of the 6th International Conference on Communication and Network Security*, 2016, pp. 104–112.

[36] M. Alidoost Nia and A. Ruiz-Martínez, “Systematic literature review on the state of the art and future research work in anonymous communications systems,” *Comput. Electr. Eng.*, vol. 69, pp. 497–520, 2018.

[37] M. Naor and B. Pinkas, “Efficient oblivious transfer protocols,” *Proc. Annu. ACM-SIAM Symp. Discret. Algorithms*, pp. 448–457, 2001.

[38] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, “More Efficient Oblivious Transfer Extensions,” *J. Cryptol.*, vol. 30, no. 3, pp. 805–858, Jul. 2017.

[39] T. Chou and C. Orlandi, “The Simplest Protocol for Oblivious Transfer,” in *Progress in Cryptology -- LATINCRYPT 2015*, 2015, pp. 40–58.

[40] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, “More efficient oblivious transfer and extensions for faster secure computation,” *Proc. ACM Conf. Comput. Commun. Secur.*, pp. 535–547, 2013.

[41] I. Damgård, “Commitment Schemes and Zero-Knowledge Protocols,” *Lect. Data Secur. Mod. Cryptol. Theory Pract.*, pp. 63–86, 1999.

[42] I. Damgaard and J. B. Nielsen, “Commitment Schemes and Zero-Knowledge Protocols,” pp. 1–36, 2011.

[43] S. Matsuo and W. Ogata, “Brief Announcement: A Method for Exchanging Valuable Data: How to Realize Matching Oblivious Transfer,” *Proc. Annu. ACM Symp. Princ. Distrib. Comput.*, vol. 22, p. 201, 2003.

[44] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *International Workshop on Information Hiding*, 2002, pp. 400–414.

[45] B. Pinkas, T. Schneider, and M. Zohner, "Faster Private Set Intersection Based on OT Extension," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 797–812.

[46] E. Stark, M. Hamburg, and D. Boneh, "bitwiseshiftleft/sjcl: Stanford Javascript Crypto Library," *github.com*, 2010. [Online]. Available: https://github.com/bitwiseshiftleft/sjcl/. [Accessed: 06-Nov-2020].

[47] E. Stark, M. Hamburg, and D. Boneh, "Stanford Javascript Crypto Library," *github.io*, 2013. [Online]. Available: http://bitwiseshiftleft.github.io/sjcl/. [Accessed: 06-Nov-2020].

[48] E. Stark, M. Hamburg, and D. Boneh, "Symmetric Cryptography in Javascript," in *Annual Computer Security Applications Conference*, 2009.

[49] Web Cryptography Working Group, "Web Cryptography API," W3C, Jan. 2017.

[50] K. Cairns, H. Halpin, and G. Steel, "Security analysis of the W3C web cryptography API," in *International Conference on Research in Security Standardisation*, 2016, pp. 112–140.

[51] MDN contributor: wbamberg, "Web Crypto API - Web APIs | MDN," *MDN Web Docs*, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API. [Accessed: 06-Nov-2020].

[52] A. Deveria, "Can I use: Web Cryptography," *caniuse.com*, 2020. [Online]. Available: https://caniuse.com/cryptography. [Accessed: 09-Nov-2020].

[53] Web Cryptography Working Group, "Web Cryptography API," W3C, Nov. 2019.

[54] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digit. Investig.*, vol. 3, pp. 91–97, 2006.

[55] F. Breitinger and H. Baier, "A fuzzy hashing approach based on random sequences and hamming distance," 2012.

[56] P. A. Grassi *et al.*, "NIST SP 800-63B Digital Identity Guidelines: Authentication and Lifecycle Management," *Natl. Inst. Stand. Technol. Spec. Publ.*, vol. 800, no. 63B, pp. 1–69, 2017.

[57] M. Garcia, "Easy Ways to Build a Better P@$5w0rd," *nist.gov*. [Online]. Available: https://www.nist.gov/blogs/taking-measure/easy-ways-build-better-p5w0rd. [Accessed: 04-Dec-2020].

[58] B. Pinkas, T. Schneider, and M. Zohner, "Scalable Private Set Intersection Based on OT Extension," *ACM Trans. Priv. Secur.*, vol. 21, no. 2, Feb. 2018.

[59] M. Dworkin, "NIST SP 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," *Natl. Inst. Stand. Technol. Spec.*

Publ., vol. 800, no. 38D, pp. 1–39, 2007.

[60] Q. Dang, "NIST SP 800-107 Rev. 1 Recommendation for Applications Using Approved Hash Algorithms," *Natl. Inst. Stand. Technol. Spec. Publ.*, vol. 800, no. 107, pp. 1–25, 2012.

[61] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[62] G. Alagic *et al.*, "Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process," 2020.

[63] MDN contributor: Mathieu Deaudelin, "How to turn off form autocompletion - Web security | MDN," *MDN Web Docs*, 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Securing_your_site/Turning_off_form_autocompletion. [Accessed: 11-Feb-2021].

[64] F. Kargl *et al.*, "Strengthening Enforcement in a Comprehensive Architecture for Privacy Enforcement at Internet Websites," *Front. Comput. Sci. | www.frontiersin.org*, vol. 2, no. 2, 2020.

[65] C. Adams and K. Barbieri, "Privacy Enforcement in E-Services Environments," in *Privacy Protection for E-Services*, G. Yee, Ed. Hershey, PA, USA: IGI Global, 2006, pp. 172–202.

[66] National Institute of Standards and Technology and Canadian Centre for Cyber Security, "Implementation Guidance for FIPS 140-3 and the Cryptographic Module Validation Program," 2020.

[67] E. Barker, "NIST SP 800-57: Recommendation for key management: Part 1 – General General," *Natl. Inst. Stand. Technol. Spec. Publ.*, vol. 800, no. 57 Part 1 Revision 5, pp. 1–142, 2020.

[68] A. Jivsov, "Elliptic Curve Cryptography (ECC) in OpenPGP," RFC6637, 2012.

[69] D. J. Bernstein and T. Lange, "Security dangers of the NIST curves," *17th Workshop on Elliptic Curve Cryptography*, 2013. [Online]. Available: https://cr.yp.to/talks/2013.05.31/slides-dan+tanja-20130531-4x3.pdf.

[70] H. Al Osman, "Architecture, Web Application Framework, Design Pattern," in *EBC5380: Systems and Architectures for Electronic Commerce*, 2018.

Appendix A - Literature Review Matrix

Table 12: Literature review matrix (without work not cited in this thesis) – updated version of [3, Fig. 2]

					Research topic			
Reference	Type of work	Type of research contribution	Unit of Analysis	Analysis method	Information privacy concerns	Attitude towards e-shopping	Privacy preserving architectures	Other
Ackerman, Cranor, & Reagle, 1999 [20]	CONF	Exploratory	consumer (individual)	panel based survey of 381 US households	x	x		clustering, information privacy concern difference when children involved, automatisms for disclosure of personal information
Androulaki, 2011 [6]	THES	Design & Action (Architecture, PoC)	n/a	n/a			x	
Bansal, Zahedi, & Gefen, 2016 [19]	JOUR	Explaining & Predicting	consumer (individual)	controlled lab experiment with 367 US Students at Midwestern university, questionaire, statistical evaluation (e.g. factor analysis)	x	x		trust, extroversion, agreeableness, conscientiousness, intellect, prior experience
Belanger, Hiller, & Smith, 2002 [5]	JOUR	Explaining & Predicting	consumer (individual)	controlled presentation of selected web sites to 140 US students, questionaire, expert ratings for privacy and security features, statistical evaluation (e.g. factor analysis)	x	x		ranking of security features, security seals, privacy statements, privacy seals, pleasure features, perceived trustworthiness
Bella, Giustolisi, & Riccobene, 2011 [23]	JOUR	Design (Protocol, Function, Paradigm)	n/a	n/a			x	
Eastlick, Lotz, & Warrington, 2006 [18]	JOUR	Explaining & Predicting	consumer (individual)	questionaire based survey of 477 random US households (internet subscribers), statistical evaluation (e.g. factor analysis)	x	x		trust, perceived reputation
Groppe, Kuhr, & Coskun, 2018 [2]	JOUR	Design (Architecture)	n/a	n/a			x	
Ha & Stoel, 2009 [16]	JOUR	Explaining & Predicting	consumer (individual)	questionaire based survey of 298 US Students, statistical evaluation (e.g. factor analysis)		x		e-shopping quality, perceived ease of use, perceived trust, perceived shopping enjoyment, perceived usefulness
Li, Zhang, & Zhang, 2016 [7]	CONF	Design & Action (Architecture, PoC, Performance Tests)	n/a	n/a			x	
Limbu, Wolf, & Lunsford, 2012 [21]	JOUR	Explaining & Predicting	consumer (individual)	questionaire based survey of 240 US students, statistical evaluation (e.g. factor analysis)		x		perceived e-tailer ethics, trust
Miyazaki & Fernandez, 2001 [22]	JOUR	Explaining & Predicting	consumer (individual)	questionaire based survey of 160 adults, statistical evaluation (e.g. regression analysis)		x		internet experience, perceived risk, adoption of offline remote retail methods
P. Riquelme & Román, 2014 [17]	JOUR	Explaining & Predicting	consumer (individual)	questionaire based survey of 398 consumers (prob. Spanish), statistical evaluation (e.g. confirmatory factor analysis)		x		perceived security, perceived privacy
Rennhard, Rafaeli, Mathy, Plattner, & Hutchison, 2004 [4]	JOUR	Design (Architecture)	n/a	n/a			x	
Sasson et al., 2014 [24]	CONF	Design & Action (Scheme, Performance Tests)	n/a	n/a			x	
United States Patent No. US 2001/0029472 A1, 2001 [25]	PAT	Design (Architecture)	n/a	n/a			x	

Appendix B - Literature Review & Concept Matrix

Table 13: Literature review & concept matrix of technical designs – updated version of [3, Fig. 1]

					Anonymized phases/concepts					
Reference	Type of work	Approach	Contribution	Level of detail	Browsing/selecting goods	Specification of payment and shipping information	Payment authorization	Delivery/receiving of goods	Other	Drawbacks and limitations
Androulaki, 2011 [6]	THES	Identity Management Model	architecture, security and privacy analysis, partial proof of concept, partial performance analysis	very high	+	+	+	+	strong authentication, internet browsing, online advertisement, evaluation systems, taxable bank accounts, risk management (credit scoring), employment payment, criminal investigations	high integration effort & therefore acceptance of e-tailers, & banks
Bella, Giustolisi, & Riccobene, 2011 [23]	JOUR	usage of cover data, balancing of anonymity and trust	new paradigm, strengthened e-poll protocol, new differential-privacy preserving function	high	o	-	-	-		only anonymizes browsing of goods, afterwards not anonymous
Groppe, Kuhr, & Coskun, 2018 [2]	JOUR	Separation of Data	architecture, privacy analysis, analysis of current possibilites to stay anonymous during online shopping	low	+	+	+	+	returns (reclamation), criminal investigation	low level of detail on implemen-tation/cryptographic primitives, high integration effort, questionable handling of criminal investigations via merging of data, acceptance of e-tailers & consumers
Li, Zhang, & Zhang, 2016 [7]	CONF	Attribute based encryption & hidden access tree	privacy-preserving logistics system with fine-grained access control on data, security analysis, realistic performance analysis	high	-	o	o	+	correction of wrong delivery	low level of detail on payment for delivery, no handling of criminal investigations & returns, acceptance of e-tailers
Rennhard, Rafaeli, Mathy, Plattner, & Hutchison, 2004 [4]	JOUR	Pseudonymity components	architecture including pseudonymity network and pseudonymous secure electronic transaction protocol	high	+	o	+	-	internet browsing, delivery/receiving of electronic goods, criminal investigation	local proxy necessary, acceptance of e-tailers, banks & consumers
Sasson et al., 2014 [24]	CONF	Improvement of Bitcoin	decentralized anonymous payment schemes (DAP schemes), security proof, implementation as Zerocash, realistic performance analysis	high	-	o	+	-		Acceptance of crypto currencies, no handling of criminal investigations, acceptance of e-tailers & consumers
United States Patent No. US 2001/0029472 A1, 2001 [25]	PAT	Terminal based design, implicit separation of data	architecture	middle	+	+	o	o		payment via prepaid credit cards, delivery via pick up at convenience store, no handling of criminal investigations & returns, acceptance of e-tailers & consumers

Appendix C

Detailed Description of Prototype Development Environment

The prototype development environment implements the *GhostBuy Prototype's* architecture through 3 projects representing the *Portal Application* (which includes/provides the *Client Application* respectively the *Portal Application's client-side functionality*), the *Frontend Application,* and the *Backend Application*. To be able to run these applications on the same development-computer, each application uses a dedicated Tomcat v 9.0 server instance, with each instance running on different ports. Since the *Portal Application* is the application with which the customer directly interacts, it uses the default https port 443 (http 80 is automatically redirected to https) and locally can be browsed using the URL *www.ghostbuy.xyz*. The *Frontend Application* uses https port 61443 and can be accessed via the local URL *https://frontend.ghostbuy.xyz:61443*.[84] Finally, the *Backend Application* uses https ports 62443 and can be accessed via the local URL https://backend.ghostbuy.xyz:62443.[85] In the development environment the web applications are named according to these URLs, i.e. e.g. the *Portal Application* is named *www.ghostbuy.xyz*.

As illustrated in Figure 8, the *Frontend Application* and *Backend Application* implement the web services and database functionality that represent the *GhostBuy Prototype*'s split *Model*, where an application-specific web service client is provided to access the respective application's web services.

The *Portal Application* implements the controllers, and servlet filters that dynamically provide the HTML pages and other web resources (*CSS, JavaScript*) to the customer's web browser. Most of the *Portal Application's client-side functionality* – including the *GhostBuy* -specific cryptographic functionality that is based on the *WebCrypto API* – is used on multiple pages. To avoid having redundant code, we therefore implemented a JavaScript module that encapsulates this functionality in accordingly structured namespaces. This module, named

[84] We configured http port 61080 for the application but it is not being used.

[85] We configured http port 62080 for the application but it is not being used.

ghostbuy_module.js, exports the necessary constants and functions so that they can be used and triggered by the other – webpage-specific – JavaScript modules.

The web pages that are provided by the *Portal Application* are based on a web page template called *OneTech* that we obtained from *colorlib.com.* The template includes some supporting *JavaScript* functionality, e.g. to initialize a drop-down menu. It must be understood that we strongly modified the template's layout and implemented all non-static website functionality. I.e. the template itself only displays static placeholder content, including static product descriptions, prices, images, and even a static number of "items in cart". The only exception to this is the filtering and sorting functionality that we nevertheless needed to modify for our purposes as described in Section 5.9.

The communication between the three web applications takes place using TLSv1.2 connections. The same applies to the communication between the customer's browser and the *Portal Application*.

The *Frontend Application* and *Backend Application* can permanently store account information and order information by saving encrypted Java objects (that we dubbed *DataBase Objects (DBO)*)[86] in dedicated MySQL instances running on ports 53306 (*Frontend*) and 53307 (*Backend*). Hereby *Hibernate* has been used to implement the *object-relational mapping (ORM)*.

In addition to the three web application projects, the development environment contains two further projects in which we implemented the *DTOs (*named *ghostbuy.xyz-DTO)* and the objects providing cryptographic functionality *(*named *ghostbuy.xyz-Crypto).* These two projects, respectively the objects they implement, are used by the three web applications. I.e. they have been made available to the web application projects by adding them to their Java Build Paths. This ensured that all web application projects always used the very same (and latest) version of the DTOs and cryptographic objects.

The top-level project structure as configured in the development environment that we used for implementation (Eclipse IDE for Enterprise Java Developers 2020-03) can be seen in Figure 9.

[86] In the context of Hibernate/ORM otherwise often called "beans"

Appendix D

Detailed Description of Initial Cryptographic Setup Process

The following is a more detailed description of the *Initial Cryptographic Setup*. Unlike the less detailed description (see Section 5.5) it includes which exact component of an application performs a certain step and how each process step is triggered. Therefore, it has two process steps more, that, however, we did not picture in Figure 38:

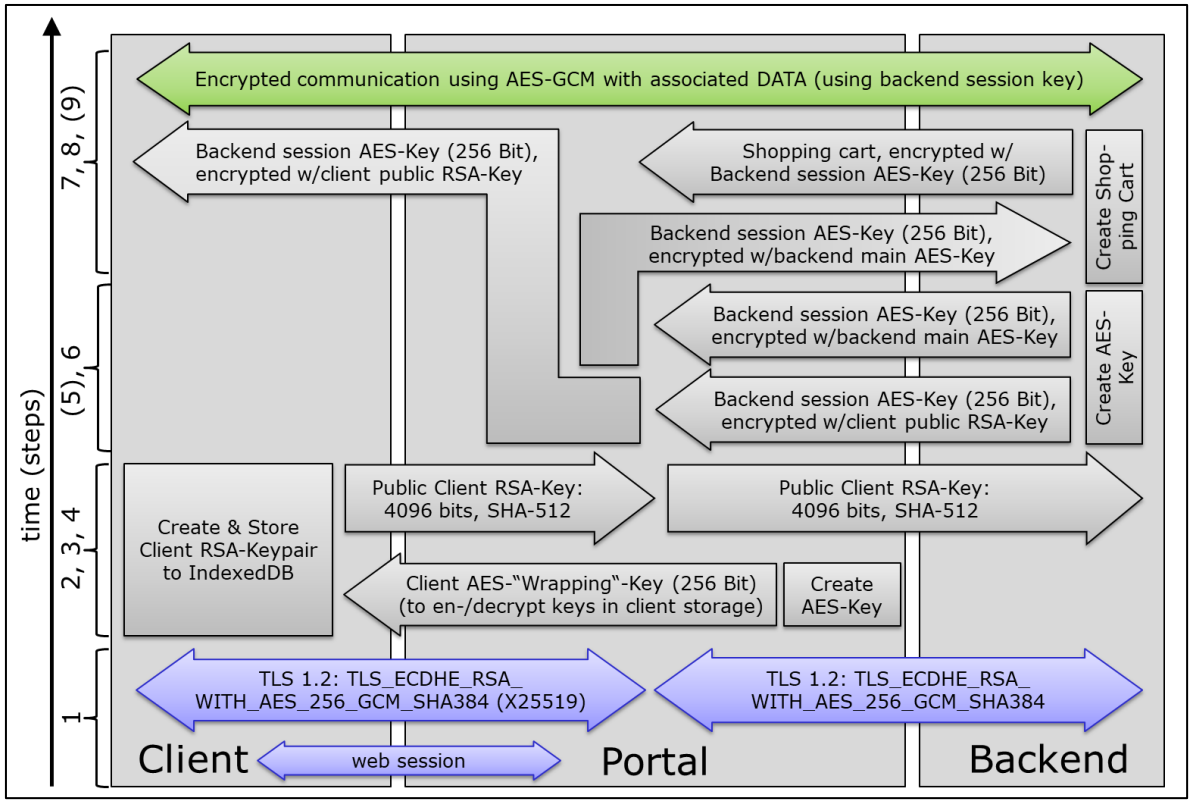

Figure 38: Initial Cryptographic Setup process (detailed)

1. The first step of the initial setup process is triggered by a customer browsing the *Portal Application's* domain *(www.ghostbuy.xyz)*. The customer's web browser establishes a secure connection to the *Portal Application's* web server. Since at the time of implementation TLSv1.3 was not fully supported by all browsers that we used for testing, the server is configured to use TLSv1.2 with ciphers *TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384* or

TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384.[87] It is important to note that the complete communication between the customer's web client (browser) and the *Portal Application* will take place over this secure connection, whereby – since the https protocol is stateless – each new request will use a new connection. The *Portal Application's* session filter intercepts the request and creates a web session. This includes assigning a web session ID that – later on – will be recorded in the customer's browser by setting a session cookie. As explained in the following, that means that the client only will be able to use the keys and other information that is part of the *Initial Cryptographic Setup* as long as the web session remains active on the server side and the client has access to the session cookie. Most tested browsers – with the exception of Opera – by default delete a session cookie when they are closed so that closing the browser effectively blocks any subsequent user of the browser from accessing/continuing the web session. The session filter then redirects the request to the so-called *SessionAndCryptoInitializationController.*

2. The *SessionAndCryptoInitializationController* controller creates an AES256 key and stories it in the web session object in encrypted form.[88] This key, that we name *Client Wrapping Key (2-P/C),* will be used for encrypting and decrypting sensitive session-related information that is stored in the customer's browser-local *IndexedDB*. In addition, the current initialization state of the client is saved to the web session, too, in order to be able to correctly detect the client's initialization status during the following steps: Next, the *SessionAndCryptoInitializationController* forwards the request to a special initialization web page that will trigger the next initialization steps.
3. When the initialization web page is loaded by the customer's browser, an initialization script (JavaScript) is being executed. The script uses an AJAX request to the *Portal Application's GetSessionObjectController* to obtain the client's current initialization status: Subsequently, it creates an asymmetric key pair (*RSA-OAEP* with 4096 bits, using SHA-512 as hash function) which it saves to the browser-local *IndexedDB* (in

[87] These are the two most secure ciphers at time of implementation of which at least one is supported by all tested browsers.

[88] For encryption it uses a *Portal Application*-specific AES256 key that we call *Portal Application Controller Key,* see *Key ID* 1-P in Table 4.

encrypted from). For encryption (and later decryption) it uses the AES256 key that was created by the server in the previous step, hence the name *Client Wrapping Key (2-P/C)*. To obtain the *Client Wrapping Key (2-P/C)* it performs an AJAX request (to the *Portal Application's GetSessionObjectController).* Since this and other requests are implicitly authenticated via the web session ID, the key can only be requested from within this browser and only as long as the web session is active. To recall, although the key is called *Client Wrapping Key (2-P/C)* it is also used to encrypt/decrypt all other sensitive session-related information in the browser-local *IndexedDB*.

4. After saving the RSA key pair, the initialization script sends the public key, the *Client RSA Public Key* (*3-C/B*), to the *Portal Application* via the so-called *SetSessionObjectController.* The key is saved to the server-side web session object (again, in encrypted form, although this is not absolutely necessary since it is a public key). Additionally, the current initialization state of the client is saved to the web session.
5. After having sent the public RSA key to the server the initialization script reloads the current web page. Since the client is not yet fully initialized the session filter redirects the request to the *SessionAndCryptoInitializationController* again.
6. The *SessionAndCryptoInitializationController* now triggers the so-called *BackendCryptoInitializationService* at the *Backend Application* to create another AES256 key. This key, that we named *Backend Session Key (6-B/C),* is returned in two separately encrypted forms. One time it is encrypted with the *Client RSA Public Key* (*3-C/B*) (which was provided by the *SessionAndCryptoInitializationController* in the request to the *BackendCryptoInitializationService).* A second time it is encrypted with an AES256 key that is shared by all *Backend Application* web services and therefore called *Backend Application Web Service Key* (*7-B*). Both forms of the key are saved to the web session which is managed by the *Portal Application* and therefore accessible to the *SessionAndCryptoInitializationController*. Due to this, the *Backend Application* does not need to keep track of the web sessions. The *Portal Application* will provide the encrypted *Backend Session Key (6-B/C)* [89] with each subsequent

[89] Encrypted with *Backend Application Web Service Key* (*Key ID 7-B).*

(encrypted) request on behalf of the client. The *Backend Application* only needs to decrypt the key in order to be able to decrypt the request of the client. We call this technique *Virtual Session,* hence the name *"Backend Session Key" (6-B/C)*.

7. After the two forms of the *Backend Session Key (6-B/C)* have been saved, the *SessionAndCryptoInitializationController* requests the *Backend Application's ShoppingCartProcessService* to create a new shopping cart. As outlined in the previous step, for this it provides the encrypted *Backend Session Key (6-B/C)* [90] so that the *Backend Application's ShoppingCartProcessService* can use it to encrypt the *Shopping Cart Collection.* When the *SessionAndCryptoInitializationController* received the *Shopping Cart Collection* (that it cannot decrypt), it saves it to the web session (i.e. it becomes part of the *Virtual Session*).

   After the encrypted *Shopping Cart Collection* has been saved to the web/virtual session, the *SessionAndCryptoInitializationController* forwards the request again to the initialization web page causing a transparent page reload that will trigger the last initialization step.
8. When the initialization web page is reloaded, the initialization script (JavaScript) is being executed once more. The script obtains the current initialization status from the *Portal Application's GetSessionObjectController,* again. Subsequently it requests the RSA-encrypted *Backend Session Key (6-B/C)* (that was saved to web session in the previous step) from the same controller and decrypts it using its private RSA key. To do so it has to retrieve the private RSA key from the IndexedDB and decrypt it using the *Client Wrapping Key* that it will request from the *GetSessionObjectController,* again. Since the client now has obtained all necessary information, the *Portal Application's GetSessionObjectController* will save this initialization status to the web session. At the client-side the script saves the *Backend Session Key (6-B/C) to the IndexedDB* (re-encrypted with the *Client Wrapping Key (2-P/C)*) and afterwards reloads the current web page a last time.
9. Since the client now is fully initialized the session filter redirects the request to the *Portal Application's* home page.

[90] Encrypted with *Backend Application Web Service Key* (*Key ID 7-B).*

When the *Initial Cryptographic Setup* has been built, all further communication between the client and the *Backend Application* is encrypted with the *Backend Session Key (6-B/C),* i.e. the *Portal Application* (and the *Frontend Application*) cannot read the content of this communication although it de-facto defines the structure of the exchanged encrypted information: To recall, in the *GhostBuy Prototype* the *Client Application* executes JavaScript code that is being provided by the *Portal Application*.

The complete process of building the *Initial Cryptographic Setup*, as described above, takes only about 2 to 5 seconds to complete[91] although, as described, it involves many different controllers, web filters, web services and requires multiple – transparent – web page forwards and redirects. Meanwhile the customer sees a loading animation that reads "Building cryptographic setup". The process is designed in a way that it can be resumed or resumes automatically in the unlikely case that e.g. the customer manually interrupts it or retriggers it before completion.

The process could presumably even be sped up slightly by making the initialization script's page reload unnecessary. This would require restructuring the initialization script and changing the *Portal Application*`s initialization states but in general should be no bigger challenge. However, due to a lack of time and considering the comparably small advantages this change would likely yield, we did not implement it.

Finally, we would like to point out that we argue that the usage of a *Client Wrapping Key (2-P/C)* that is stored and provided by the *Portal Application* further mitigates the risk of so-called *Key Exchange API Attacks* on cryptographic keys used by the *Web Crypto API* (for details, see Section 6.3.5).

[91] Tested using current versions of the browsers Edge (non-legacy), Chrome, Mozilla Firefox, Opera.

Appendix E

Detailed Description of Dynamically Adding Encrypted Content

The following is a more detailed description of the process that dynamically adds encrypted content to a web page of the *GhostBuy Prototype.* Unlike the less detailed description (see Section 5.6.2) it includes which exact component of an application performs a certain step and how each process step is triggered:

1 & 2: When the home page has been loaded by the browser, a web-page specific script is being executed that – via invocation of subordinate functions which are implemented in the JavaScript module *ghostbuy_module.js* – makes an AJAX request (*https-get*) to obtain the shop list from the Portal Application's so-called *GetPageContentController* .

3 & 4: The *GetPageContentController* calls the *Backend Application*'s *ProductCatalogService's* method called *"getShopList"* (*https://backend.ghostbuy.xyz:62443/webservices/ProductCatalogService/getShopList*). To do so the *GetPageContentController* invokes the *BackendWebServiceClient's method "*getShopList" (for obvious reasons named the same as the web service method) which encapsulates the call including the interpretation of the returned status code(s) and conversion of the encrypted shop list to an encrypted java object[92]. In order to enable the *Backend Application's ProductCatalogService* to encrypt the shop list with the client-specific *Backend Session Key (6-B/C)* the *BackendWebServiceClient's* accepts a so-called *ClientRequestDTO* in which the *GetPageContentController* passes on the *Backend Session Key (6-B/C)* which has been encrypted with the *Backend Application Web Service Key* (7-B*)* (see step 5 of the *Initial Cryptographic Setup* described in Section 5.5).

[92] The web service itself returns a textual representation of the encrypted object using *JavaScript Object Notation (JSON)*

5, 6, 7: The *ProductCatalogService* decrypts the *Backend Session Key (6-B/C)* and uses it to encrypt the shop list[93] to create a so-called *AesGcmEncryptedDataUnsignedDTO* which it then returns to the *GetPageContentController* (via the *BackendWebServiceClient*). The *GetPageContentController* – who is not able to decrypt the content of the DTO – returns it to the calling function in the web page's JavaScript execution with the script receiving a JSON representation of the DTO.

Finally, the *Client Application* decrypts the content – for which it has to restore and decrypt the *Backend Session Key (6-B/C)* from the browser-local *IndexedDB* – using the *Web Crypto API*. It then manipulates the web page's *Document Object Model (DOM)* to populate the shop selector drop-down box with the list. Please see Appendix M for an illustration of the main steps of the decryption and conversion process as performed in the *Client Application*.

[93] For performance reasons, the shop list is pre-generated at startup of the *ProductCatalogService* via iterating through all so-called *ECommercePlatformConnectors* (for the prototype we only implemented one *ECommercePlatformConnector,* namely the *EbayConnector*) using the API implemented therein to get the list of shops supported by the respective platform.

Appendix F

Detailed Description of Client-Side Parameter Encryption and Submission

We achieve this by encrypting search terms and other parameters at the client-side using the *Backend Session Key (6-B/C)*. In the prototype, the encryption in the *Client Application* is performed by JavaScript functions that utilize the *Web Crypto API*. Apart from that the process is almost identical to the example from the previous appendix: The only two differences are, firstly, that the content is requested using an AJAX request that uses *https-post* so that the encrypted parameters can be sent to the *GetPageContentController* in the request body. Secondly, the *GetPageContentController* adds the parameters to the *ClientRequestDTO* so that the *Backend Application's ProductCatalogService* can decrypt and use them. Additionally, the data flow does not end here, but the *ProductCatalogService* uses what we call an *ECommercePlatformConnector* (e.g. *EbayConnector*) to obtain the requested product information from the respective e-commerce website.

Appendix G

Regarding the usage of https-post for "simple" content requests

It is common knowledge that *https-post* in general is intended to be used for submitting to-be-processed data and for creating or updating resources. In contrast to this *https-get* should be used for requesting data. At first glance, our requirements and implementation do not seem to fit this definition, but while it is not of great importance, they actually might for the following two reasons: Firstly, the encrypted parameters need to be processed – at least more than e.g. a simple plaintext search. Secondly, the response depends on a resource that was stored on the server, namely the *Backend Session Key (6-B/C),* but it is unique for each request – even when the same request parameters where specified – since each response is encrypted with a different initialisation vector. I.e., the server creates a new resource for each request, but the resource is not stored on the server.

Appendix H

Detailed Description of Sign-In Process Steps

1. When a customer needs to or wants to sign in, he/she is taken to the *Sign-In Page*.
2. On the *Sign-In Page* the customer must specify their account name and password.
3. When the customer clicks the "Sign In"-button the following process is triggered: The *Client Application* requests the account name hashing parameter pair – which in general is the same for all customers – from the *Portal Application* and hashes the account name using the PBKDF2 algorithm with SHA-512 hash function.
4. Next, the *Client Application* requests the password hashing parameters pair from the *Portal Application,* sending the account name hash as a parameter. This is to check if the specified account exists and to obtain the password hashing parameters that were specifically generated and saved for the respective account's password during the sign-up process. For this, the *Portal Application* rehashes the account name hash with the same rehashing parameters as during the sign-up process (that it retrieves from the *Frontend Application's* database) and then requests the *Frontend Application* to retrieve the password hashing parameters for the account stored under this hash. If the account does not exist, an according status and message is returned to the *Portal Application* and displayed in the *Client Application.* Otherwise, the password hashing parameters are returned.
5. After having received the password hashing parameters, the *Client Application* hashes the password (using PBKDF2 with SHA-512). The hash is saved for step 8.
6. Now, the *Client Application* requests the key hashing parameters pair from the *Portal Application*, sending the account name hash as a parameter. The process is equivalent to step 4. If the account would not be found – which in general should not happen since its existence has already been checked in step 4 – an according status and message is returned to the *Portal Application* and displayed in the *Client Application.* Otherwise, the key hashing parameters are returned.[94]

[94] This step and step 4 could be merged into one step, i.e. both hashing parameter pairs could be retrieved at the same time. We implemented separate steps since otherwise we would have had to create another custom DTO.

7. After having received the key hashing parameters, the *Client Application* computes the *Client-side Secret Key (5-C)* (using PBKDF2 with SHA-512) and saves it to the *IndexedDB* (encrypted with the *Client Wrapping Key (2-P/C)* that it obtains from the *Portal Application*). Since this key is the same as the key that has been generated during the sign-up process it will be used, if applicable, to encrypt the *Backend Session Key (6-B/C)* when an order is completed and to decrypt it when a previous order is opened in the *Order History Page* (see Section 5.6.9).
8. After that, the *Client Application* sends the account name hash and hashing parameters, the password hash and hashing parameters to the *Portal Application* for verification.
9. The *Portal Application* double-checks the received data (e.g. regarding empty strings) and – if everything is OK – rehashes the account name hash. The account name rehashing parameters are retrieved from the *Frontend Application's* database as before. Then, the *Portal Application* retrieves the password rehashing parameters from the *Frontend Application`s* database, too, using the rehashed account name hash to identify the respective account record. The *Portal Application* uses the retrieved parameters to rehash the password hash. Subsequently, the *Portal Application* retrieves the account object from the *Frontend Application's* database (using the account name rehash for lookup).[95]
10. Now, the *Portal Application* compares the computed rehashed password to the rehashed password that was stored in the account object. Hereby we use a constant-time comparison, to mitigate the risk of timing attacks. If the password (re-)hashes do not match an according status and message is returned to the *Portal Application* and displayed in the *Client Application.* Otherwise, the *Portal Application* removes unnecessary and/or sensitive information such as the hashing parameters and the password hash itself from the account object and saves it to the session (encrypted). As during the sign-up process, it also records the authenticated state in the session so that the customer sees new header menu options ("My orders" = view order history &

[95] The password rehashing parameters (server-side) also could be retrieved from the account object, hereby reducing the number of database requests.

"Sign out") and also is granted access to pages which require authentication, e.g. the *Checkout Page* and the *Order History Page.*

Appendix I

Detailed Description of State Preservation Process Steps

In detail, our solution for preserving the client state – regarding selected parameters – between page loads and to allow for browser (history) navigation is described by the steps below – where the transition between *Home Page* and *Shop Page* should be seen as an example for all transitions. Figure 27 illustrates the steps. Please note that all steps are performed by the *Client Application* (unless stated otherwise), i.e. – in our prototype – by executing JavaScript code in the browser:

1. When a customer-chosen set of parameters (e.g. search terms, selected shop, pagination size) must be submitted to the *Backend Application* and/or must be preserved between two page loads (e.g. the currently selected shop is also preserved when opening the order history page but not submitted to the Backend) these parameters are converted into a JSON object which is encrypted with the *Backend Session Key (6-B/C)*. The result – that is a set of encrypted query parameters which we call *AesGcmEncryptedQueryJson* – is kept in memory for step 3.
2. Next, a string-representation of the parameters is built by concatenating their textual representation (e.g. text "100" instead of number 100), with a delimiter (",") being placed between each parameter (to avoid that strings match "by chance", e.g. "10" & "11" otherwise equals "101" & "1").[96] The string is hashed using the SHA-256 hashing algorithm with a session-specific secret 256 bit salt being added. The salt is generated on first use and stored in the *IndexedDB* (encrypted with the *Client Wrapping Key*) so that it can be reused for generating subsequent hashes. The resulting hash is called *URL-ID,* since it will be used as the identifier in the URL (and *IndexedDB*).

[96] We did not use a textual representation of the JSON object built in step 1, since we did not want to rely on that the converted string would always be the same as we have no control over the JSON object to JSON string conversion process.

3. Now, the *AesGcmEncryptedQueryJson* (built in step 1) is stored in the *IndexedDB* using the *URL -ID* as identifier for the DB record.[97]
4. After successfully saving the parameter set to the IndexedDB the target page is being opened, with the *URL-ID* being passed as an URL parameter (note that it is not being used by the *Portal Application's* controller but by the *Client Application* in the next step).
5. When the target page has been loaded by the browser the *Client Application* (in the prototype i.e. JavaScript executed in the browser) extracts the URL-ID *parameter* from the URL and restores the associated encrypted set of parameters. If applicable, it uses it (still encrypted) to request the dynamic content that should be loaded into the page. This is possible since the parameters had been encrypted with the *Backend Session Key (6-B/C)* (not the *Client Wrapping Key (2-P/C)*). However, it also decrypts the parameter set to retrieve the chosen parameters and e.g. to populate the search box with the search terms or to highlight the selected product category. On some pages (e.g. the home page) this is the only purpose of the encrypted parameter set, since they do not load content based on customer-chosen parameters.

[97] Additionally, the ID (not URL-ID) of the target page is saved (e.g. "Home", "Shop" or "Product") since that defines the context of the encryption which is used as *Associated Data* in the AES-GCM encryption.

Appendix J

Detailed Description of Sign-Up Process Steps

Before a customer can sign up or sign in, the following must be performed once: A special function of the *Frontend Application* is executed to add two pairs of hashing parameters to the *Frontend Application's* database. Each pair consists of a hash salt and a hash iteration counter. One pair will be requested and used by the *Client Application* (client-side hashing of account name) in step 3 and the other pair will be requested and used by the *Frontend Application* (server-side rehashing of account name hash) in step 6. In general, these two pairs of hashing parameters stay unchanged, i.e. they are the same for all user account hashes to allow for efficient account lookups and retrievals in and from the database. The respective steps, 3 and 6, will give additional detail.

The sign-up process' steps are:

1. When a customer has to or wants to sign in, he/she is taken to the *Sign-In Page* first from where the *Sign-Up Page* can be opened.
2. On the *Sign-Up Page* the customer must choose an account name and specify their first and last name and optionally the email address. Most significantly, however, the customer must generate a password and re-enter it to verify that it has been correctly read by them. It is assumed that the customer saves the password in a secure location, e.g. a password safe. Generating a password for the customer is not very common: Our personal experience is that web pages in general let the user choose a password, usually requiring that the password has a certain minimum length and follows composition rules e.g. regarding the usage of letters, numbers and special characters. According to Garcia [57] one downside of this approach is that this often leads to that users choose passwords that are easy to guess by a computer but hard to remember by humans. E.g. users often modify common words by substituting letters with numbers.[57] It goes without saying that a platform like *GhostBuy*, that has a special focus on ensuring the privacy and anonymity of its customers, should seek to ensure that its customers use strong passwords: As an

example, passwords that consist of the first letters of the words in a sentence[98] are easier to memorize and harder to guess, since they are not derived from a single word using a more or less foreseeable pattern. Yet, it is not possible to enforce the use of good passwords only by requiring adherence to the above composition rules and, moreover, these rules even limit the number of possible passwords. Consequently, the NIST guidelines recommend that no rules like this are enforced.[56, Sec. 5.1.1.2]

From a user perspective, Garcia proposes to use passphrases, i.e. passwords consisting of multiple words which can be easily memorized due to association but cannot be easily guessed: E.g. they should not form a semantically complete sentence and should not be the names of the user's kids.[57] The problem is that even if the respective authentication system requires to enter several words (adding up to a certain length) it cannot be ensured that e.g. these are *not* the names of the user`s kids or pets. Also, from our personal experience, users are not – yet – used to using passphrases so it might be problematic to require them to choose and enter a safe passphrase. Therefore, our implementation generates random 4-word-passphrases from a list of 42,310 nouns. This is performed within the *Client Application*, using cryptographically secure numbers that we generate via invocation of the *Web Crypto API*.[99] We argue, that we hereby achieve a good compromise between security and usability: The passphrases are guaranteed to have at least 61 bits of entropy[100] (which could be easily increased by using longer passphrases or a larger word database) and can be entered and memorized with reasonable effort even if no password safe is used. Figure 28 shows an example of a password generated with this solution.

Please note that as of now only few browsers natively allow for unmasking password fields, which, however, is required to read and re-enter the generated password and also

---

[98] E.g.: Wioaghoth?Awo! = What is orange and goes hiking on the hill? – A wandarine orange!

[99] To choose the words we derive random numbers from random keys generated with the Web Crypto API. The API's random number generator itself is discouraged to be used for key (=password) generation, despite being a cryptographically secure random number generator (CSRNG), see https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues.

[100] $2^{61} < 42{,}310^4 < 2^{62}$ – Hereby, we assume that the Web Crypto APIs key generating function – that we use to generate the random numbers – uses a sufficiently random entropy source.

is recommended by NIST for entering password [56, Sec. 5.1.1.2]. Therefore, we implemented a JavaScript and CSS based unmask-control that is browser-independent (see Figure 28). We also suppressed the native browser functionality (that for example could be used in Microsoft Edge), to avoid overlapping – and therefore confusing or unusable – unmask-controls.

3. When the customer clicks "Create account" the *Client Application* performs an evaluation of the input, e.g. regarding the matching of the password and the password verifier. If it succeeds, the following process is triggered: The *Client Application* requests a hashing parameter set from the *Portal Application*, which contains 3 hashing parameters pairs, each consisting of a hash salt and a hash iteration counter. The first parameter pair is used to hash the account name, the second parameter pair is used to hash the password (and the password-verifier) and the last pair is used to generate a *Client-side Secret Key (5-C)* from the password. Each hashing is performed using the PBKDF2 algorithm with SHA-512 hash function.

   Please note that the first pair of hashing parameters, i.e. the account name hashing parameters, is retrieved from the *Frontend Application's* database (via a request to the *Portal Application*) and in general is the same for all customers: Hashing the account name has the goal of hiding the original account name from the *GhostBuy Prototype* – it is not meant to provide additional security. However, the respectively used pair of account name hashing parameters in the *Frontend Application's* database is linked to the generated account so that these parameters could be changed if necessary.[101] The other two hashing parameter pairs, i.e. the password hashing parameters and the key generation/hashing parameters are randomly generated (only the salt, the iteration count is pre-configured) for each *Sign-Up* request and saved when the *Sign-Up* completes – see next steps.
4. Next, the *Client Application* saves the *Client-side Secret Key (5-C)* to the *IndexedDB* (encrypted with the *Client Wrapping Key (2-P/C)* that it obtains from the *Portal Application*). This key is used to encrypt the *Backend Session Key (6-B/C)* when an order

[101] In this case, during *Sign-In*, the *Portal Application* would need to provide all previously used sets of account name hashing parameters and the Client Application would need to use them to generate the account name hash until a match is found in the DB. This process could be performed in iterations.

is completed and to decrypt it when a previous order is opened in the *Order History Page* (see Section 5.6.9).

5. After that, the *Client Application* sends the account name hash and hashing parameters, the password hash (and password-verifier hash) and hashing parameters and the client-side secret key hashing parameters (note: not the key itself) as well as the other user details to the *Portal Application*.
6. The *Portal Application* double-checks the received data (e.g. regarding matching password and password-verifier hashes) and – if everything is OK – rehashes the account name hash and the password hash with server-side hashing parameters. Hereby, the account name rehashing parameters are retrieved from the *Frontend Application's* database so that they are in general the same for all account names (see initial preparation and step 3). This is to later – during *Sign-In* – allow for an efficient database lookup of account name hashes to find the respective account information (including password hashing parameters). The server-side password hashing parameters are randomly generated (only the salt). The *Portal Application* now triggers the *Frontend Application* to create an account. Hereby it saves these hashes, the hashing parameters and rehashing parameters (i.e. client-side and server-side hashing parameters) and the user details to the *Frontend Application's* database.
7. When the account has been saved to the database, the *Portal Application* additionally saves it to the session (encrypted with the *Portal Application Controller Key (1-P)*). It also records the authenticated state in the session so that the customer sees new header menu options ("My orders" = view order history & "Sign out") and also is granted access to pages which require authentication, e.g. the *Checkout Page* and the *Order History Page*.

The sign-in process' steps are similar to the sign-up process' steps so that we do not describe them in detail here (please see Appendix H for a detailed description) but only point out the main differences. The first main difference is that the hashing parameters for the password and *Client-side Secret Key (5-C)* during the sign-in process are not randomly generated but retrieved from the *Frontend Application`s* database (in which they were stored during the sign-up process). The second – rather obvious – main difference is that during the sign-in process the *Portal Application* does not save the generated password hash to the *Frontend*

Application`s database but instead compares it to the previously saved password hash. This is to verify that the customer correctly entered the password, i.e. to authenticate them. To mitigate the risk of timing attacks, we thereby use a constant-time comparison function as implicitly recommended in [56, Sec. 4.3.2].

Appendix K

Detailed Description of Checkout Data Submission and Processing Steps

When the customer specifies the payment and shipping information on the *Checkout Page* (the customer must authenticate to open the page) and clicks "Continue Checkout", the following process is triggered (see Figure 29 for an illustration of the process):

1. The *Client Application* creates a query parameter object which basically only contains the information that the *Backend Application* is to convert the *Shopping Cart Collection* to a so-called *Order Cart Collection* (which includes taxes, fees, and converted currencies, if applicable).[102] The query is encrypted with the *Backend Session Key* and therefore can only be decrypted by the *Backend Application*.
2. Then, the *Client Application* builds a checkout data object which contains the payment and shipping information. The *Client Application* also adds the encrypted query parameters created in the first step to this object.
3. The other parts of the object are not encrypted but the *Client Application* submits it to the *Portal Application* over a secure connection.
4. The *Portal Application* converts the received information to a custom *data transfer object (DTO)* to which it also adds other information from the session.
   The created DTO (*Create Frontend Order Wrapper DTO*) contains the following:
   - *Shipping Information DTO* (not encrypted)
   - *Payment Information DTO* (not encrypted)
   - *Shopping Cart Collection Processing Request DTO*
     - *Shopping Cart Collection DTO*, encrypted with *Backend Application Web Service Key* (7-B*)*
     - *Client Request DTO*
       - *Backend Session Key (6-B/C),* encrypted with *Backend Application Web Service Key* (7-B*)*

[102] Please see the "Please note"-paragraph after this process description for additional information regarding the query parameters.

        - Query Parameters ("convert to order cart"), encrypted with *Backend Session Key*

5. After that, the *Portal Application* triggers a web service at the *Frontend Application* to create a so-called *Frontend Order.* For this it uses a custom frontend web service client and passes on the *Create Frontend Order Wrapper DTO* that it created in the previous step.
6. The *Frontend Application's* web service extracts the encrypted *Shopping Cart Collection Processing Request DTO* from the *Create Frontend Order Wrapper DTO* and triggers a web service at the *Backend Application* to process it. To do so it invokes a custom backend web service client, passing on the encrypted request DTO.
7. The *Backend Application's* webservice extracts and decrypts the *Shopping Cart Collection DTO*, the *Backend Session Key (6-B/C)* and the query parameters contained in the request. It then converts the *Shopping Cart Collection DTO* to an *Order Cart Collection DTO* which includes adding fees, taxes and converting currencies where necessary (please see the "Please note"-paragraph after the process description in the main body of the thesis). It proceeds with computing the hash of a textual representation (JSON) of this object to enable authorising the order later (see Section 5.6.9). Also, it creates two separately encrypted forms of the Order Cart Collection DTO (one of them also containing the hash) to finally create a so-called ***Backend*** *Order Cart Collection DTO* ("***Backend"*** to differentiate it from the Order Cart Collection DTO):
    - *Backend Order Cart Collection DTO*
        - JSON container object, encrypted with *Backend Session Key (6-B/C), containing*
            - JSON string representation of *Order Cart Collection DTO*
            - Hashing parameters
        - *Order Cart Collection DTO,* encrypted with *Backend Application Web Service Key* (7-B*)*
        - *Backend Order Cart* total (not encrypted)
        - *Backend Order Cart* currency (not encrypted)

- *Hash of JSON string representation* of *Order Cart Collection DTO (not encrypted)*
- *Hashing parameters (not encrypted)*

Please note that the *DTO* contains the overall total in unencrypted form so that later the payment can be processed by the *Portal Application & Frontend Application.*

8. The *Frontend Application's* web service receives the *Backend Order Cart Collection* DTO and converts it to a so-called *Frontend Order DTO* to which it also adds the unencrypted shipping and payment information. Then, it returns the *Frontend Order DTO* to the *Portal Application*.
9. The *Portal Application* receives the *Frontend Order DTO* and saves it to the session (encrypted with a *Portal-Application*-specific key).[103] Additionally, it extracts and saves the encrypted JSON container object (which contains the *Order Cart Collection DTO* JSON string and hashing parameters) to the session – without encrypting these again (it is encrypted with the *Backend Session Key (6-B/C)*, i.e. it cannot be decrypted by the *Portal Application*). This is done so that it can be efficiently returned to the *Client Application*, later.
10. At last, the *Portal Application* returns a success status and the target URL (here: https://www.ghostbuy.xyz/PlaceOrder) to the *Client Application*.
11. The *Client Application* opens the target URL (https://www.ghostbuy.xyz/PlaceOrder) to display the *Place Order Page*.

[103] It also adds the customer's account ID – which it extracts from the encrypted account object that is stored in the session (see Section 5.6.7) – to the DTO.

Appendix L

Detailed Descriptions of Order Data Merging and Order Placement Process

When the *Client Application* opens the *Place Order Page* it triggers the following process – hereby continuing the process described in the previous section:

1. At first the *Client Application* obtains the shipping and payment information from the *Portal Application* via an AJAX request. The *Portal Application* restores the information from the web session, decrypts it and returns it to the *Client Application* (unencrypted, but over a secure connection).
2. Next, the *Client Application* obtains the JSON container object (containing the *Order Cart Collection DTO* JSON string and hashing parameters) from the *Portal Application* via an AJAX request, too. The *Portal Application* restores this object, which is encrypted with the *Backend Session Key (6-B/C)*, from the web session and returns it to the *Client Application*.
3. Then, the *Client Application* populates the *Place Order Page* with the shipping information, payment information and the *Order Cart Collection* information. To do the latter, is decrypts the JSON container object and converts the contained *Order Cart Collection DTO* JSON string to a JSON object. The reason for transmitting this object as a JSON string is that by doing so the *Backend Application* and the *Client Application* independently can hash the string and thus the JSON object that it represents. If it was transmitted as a JSON object we assume it could happen that the hashable textual representation at the *Backend Application* and *Client Application* would not be the same, therefore not yielding the same hashes.
4. The *Client Application* also hashes the *Order Cart Collection DTO* JSON string using the hashing parameters from the decrypted JSON container object. This hash will later be used to authorize placing the order.

When the customer clicks the “Place your order” button, the following process is triggered:

1. The *Client Application* restores the *Backend Session Key (6-B/C)* from the IndexedDB (for which it obtains the *Client Wrapping Key (2-P/C)* from the *Portal*

Application via an AJAX request). Then, it encrypts the *Backend Session Key (6-B/C)* with the *Client-side Secret Key (5-C)* that it also restores from the *IndexedDB*.

2. Subsequently, the *Client Application* submits the encrypted *Backend Session Key (6-B/C)* and the previously generated hash of the *Order Cart Collection DTO* JSON string (see last step of previous process description) to the *Portal Application*.
3. The Portal Application compares the hash computed by the *Client Application* to the hash which previously was computed by the *Backend Application* (and is stored in the encrypted *Frontend Order DTO* in the session). If they do not match this likely means that the customer updated the *Order Cart Collection* by performing a checkout process in a different browser tab. In this case the customer will be advised to refresh the *Place Order Page* to see/review the updated *Order Cart Collection* information. If the hashes match the order successfully was authorized by the customer and will be (tried to be) placed subsequently:
4. The *Portal Application* adds the encrypted *Backend Session Key (6-B/C)* to the decrypted *Frontend Order DTO* (restored from session). Due to this the customer later will be able to decrypt the Frontend Order DTO – when reviewing it in the *Order History Page* – since the client-side secret key can be regenerated and used to decrypt the *Backend Session Key (6-B/C)*. The customer otherwise would not be able to do this, since when the order successfully is placed, the session will be reinitialized and the current *Backend Session Key* therefore will be lost.
5. The *Portal Application* subsequently triggers a web service of the Frontend Application to place the order. For this it invokes the custom *Frontend Application* web service client and passes on the *Frontend Order DTO*.
6. The *Frontend Application's* web service will accept the payment information unless – for demonstration purposes – the credit card security number matches 2020. In the reject-case an according status and message would be passed back and shown in the *Client Application* (see *Order Failed Page* in Section 5.3). Please note that in a real-world implementation, the payment e.g. would be processed by invoking a third-party web service.
7. If the payment is accepted, the *Frontend Application* saves the *Frontend Order* to its database (as encrypted *Database Object (DBO)*, see Section 5.8). The hereby

assigned *Frontend Order ID*[104] is added to the respective *Account DBO* (encrypted) so that later all orders belonging to that account can be retrieved. The *Frontend Order ID* also is used as a reference ID to link the *Frontend Order* to the *Backend Order* which will be triggered to actually order the respective products at the e-commerce website (e.g. eBay). We also considered using one way-functions to create this link but decided against this since e.g. this would require using multiple key columns in the databases and secret components for these functions which – if ever changed – would make updating all references, necessary. Overall, the one-way-function-approach seemed much more complex and error-prone than the following, implemented approach which is very similar to the "Virtual Session" approach described in Section 5.4:

As mentioned above, the *Frontend Order ID* is used as reference ID to link the *Frontend Order* to the associated *Backend Order*. Vice versa, the so-called *Backend Order ID* is used as reference ID to link the *Backend Order* to the original *Frontend Order*. To protect the IDs – and thus the link – from unauthorized access, each ID is encrypted by the originating application and then stored inside/alongside[105] the respective other order. The next steps describe how this works:

8. After the *Frontend Application's* web service saved the *Frontend Order* to the Frontend Application's database – and hereby obtained the *Frontend Order ID* – it encrypts this ID with the *Frontend Application Web Service Key (9-F)*.
9. Afterwards, it submits the encrypted ID and the Frontend Order's Order Cart Collection (which is encrypted with the *Backend Session Key (6-B/C)*) to a web service at the *Backend Application* in order to trigger the respective order.
10. The *Backend Application*'s web service decrypts the *Order Cart Collection* and converts it to a so-called *Backend Order DTO*. It saves the DTO – alongside with the encrypted *Frontend Order ID* – as a so-called *Backend Order Database Object (DBO)* to its own database. Hereby all the information (the DTO and the ID) is

[104] Assigned by the invoked *Data Access Object (DAO) Implementation*

[105] The respective *DAO Implementations* save the order DTOs and IDs as encrypted Java objects as so-called DataBase Objects (DBOs).

encrypted with a *Backend Application DAO Implementation Key (8-B)*, i.e. the already *encrypted Frontend Order ID* is encrypted a second time. Due to this the original encrypted Frontend Order ID (only encrypted by Frontend) cannot be found in the database other than by decrypting the DBO – hereby avoiding leaking information regarding the linking of the orders. As a result of saving the Backend Order to its database the *Backend Application's* web service obtains the *Backend Order ID*.

11. The *Backend Application's* web service encrypts the *Backend Order ID* with the *Backend Application Web Service Key* (*7-B*) and returns this to the *Frontend Application's* web service.
12. Next, the Frontend Application's web service updates the previously created *Backend Order DBO* to include the encrypted *Backend Order ID* (which is re-encrypted with the *Frontend Application DAO Implementation Key (10-F)).* If this operation succeeds it returns an according status and the so-called *Frontend Tracking ID*[106] to the *Portal Application* which will return the to be opened URL (*Order Success Page*) to the *Client Application.*
13. The *Client Application* opens the URL of the *Order Success Page* which also shows the *Frontend Tracking ID*. Meanwhile the customer's session is invalidated, i.e. the customer is signed/logged out. This ensures that the Backend Application cannot – without further ado – attribute subsequent searches, purchases etc. to the same (unknown) customer (see Section 6.2.2).

[106] This ID is used as a temporary reference ID to find an active Frontend Order in the database.

Appendix M

Illustration of AES-GCM Decryption and Conversion Process

The following illustrates the most important steps of the AES-GCM decryption and conversion that takes place in the *Client Application,* with the encrypted shop list being considered in this example. Please note that all integers are unsigned integers since the *Web Crypto API* uses so-called *UInt8Arrays* (unsigned 8 bit integers) as input and output parameters of its methods:

The *Client Application* requests the encrypted shop list by performing an AJAX call to the URL "https://www.ghostbuy.xyz/GetPageContent?content=ShopList" (*Portal Application's GetPageContentController*) and obtains a JSON object:


```
objectJson:
    • encryptedData: (204) [107, 124, 45, 179, 119, 142, 227, 36, 49, 217, 166, 125, 161,
      243, 147, 30, 185, 20, 60, 196, 63, 129, 22, 244, 200, 254, 216, 209, 19, 112, 216,
      54, 243, 185, 222, 147, 249, 55, 112, 232, 46, 137, 100, 1, 243, 216, 217, 54, 110,
      3, 107, 141, 72, 0, 78, 43, 183, 101, 126, 110, 88, 63, 159, 172, 51, 224, 20, 172,
      191, 252, 87, 140, 240, 172, 154, 127, 149, 11, 253, 21, 40, 167, 181, 69, 57, 234,
      65, 117, 173, 83, 35, 37, 247, 4, 152, 34, 16, 7, 241, 248, 58, 140, 156, 58, 67,
      91, 26, 33, 20, 222, 41, 45, 209, 238, 116, 72, 251, 25, 23, 249, 240, 141, 74, 6,
      219, 216, 167, 109, 248, 235, 144, 204, 28, 240, 246, 135, 194, 19, 48, 195, 216,
      192, 198, 11, 113, 84, 3, 137, 184, 58, 120, 93, 27, 49, 230, 165, 172, 95, 177,
      164, 10, 36, 0, 229, 96, 130, 67, 120, 76, 33, 216, 14, 163, 109, 249, 218, 114,
      183, 151, 78, 233, 121, 178, 8, 114, 25, 29, 151, 115, 122, 31, 253, 96, 139, 209,
      92, 81, 72, 164, 218, 237, 251, 118, 216]
    • iV: (12) [194, 48, 190, 201, 188, 100, 87, 154, 141, 193, 253, 18]
```


Then, the *Client Application* restores the *Backend Session Key (6-B/C)* from the *IndexedDB,* decrypts it – for which it has to obtain the *Client Wrapping Key (2-P/C)* from the *Portal Application* – and uses it to decrypt the object. In this example the *Backend Session Key (6-B/C)* is:


```
[234, 66, 233, 100, 249, 65, 101, 255, 207, 46, 222, 83, 75, 15, 236, 2, 204, 166, 202, 200,
60, 54, 94, 157, 227, 51, 228, 97, 137, 162, 210, 196]
```


Additionally, the associated data needed for decryption is the UTF8-encoded byte representation of the string "ShopList", which is:


```
[83, 104, 111, 112, 76, 105, 115, 116]
```


The decrypted result is an array of integers which are UTF8-encoded plaintext characters:


```
pageContentJsonBytes: Uint8Array(188) [91, 34, 101, 66, 97, 121, 32, 65, 117, 115, 116, 114,
97, 108, 105, 97, 34, 44, 34, 101, 66, 97, 121, 32, 67, 97, 110, 97, 100, 97, 32, 40, 101,
110, 41, 34, 44, 34, 101, 66, 97, 121, 32, 67, 97, 110, 97, 100, 97, 32, 40, 102, 114, 41,
```

34, 44, 34, 101, 66, 97, 121, 32, 70, 114, 97, 110, 99, 101, 34, 44, 34, 101, 66, 97, 121, 32, 71, 101, 114, 109, 97, 110, 121, 34, 44, 34, 101, 66, 97, 121, 32, 72, 111, 110, 103, 32, 75, 111, 110, 103, 32, 40, 98, 41, 34, 44, 34, 101, 66, 97, 121, 32, 73, 116, 97, 108, 121, 34, 44, 34, 101, 66, 97, 121, 32, 77, 111, 116, 111, 114, 115, 34, 44, 34, 101, 66, 97, 121, 32, 83, 112, 97, 105, 110, 34, 44, 34, 101, 66, 97, 121, 32, 85, 110, 105, 116, 101, 100, 32, 75, 105, 110, 100, 111, 109, 34, 44, 34, 101, 66, 97, 121, 32, 85, 110, 105, 116, 101, 100, 32, 83, 116, 97, 116, 101, 115, 34, 93]

After decoding the character, the *Client Application* obtains the following JSON string:

["eBay Australia","eBay Canada (en)","eBay Canada (fr)","eBay France","eBay Germany","eBay Hong Kong (b)","eBay Italy","eBay Motors","eBay Spain","eBay United Kindom","eBay United States"]

Finally, the *Client Application* converts the JSON string to a JSON object:

pageContentJson: Array(11)
0: "eBay Australia"
1: "eBay Canada (en)"
2: "eBay Canada (fr)"
3: "eBay France"
4: "eBay Germany"
5: "eBay Hong Kong (b)"
6: "eBay Italy"
7: "eBay Motors"
8: "eBay Spain"
9: "eBay United Kindom"
10: "eBay United States"

Appendix N

Backend Application Database Schema

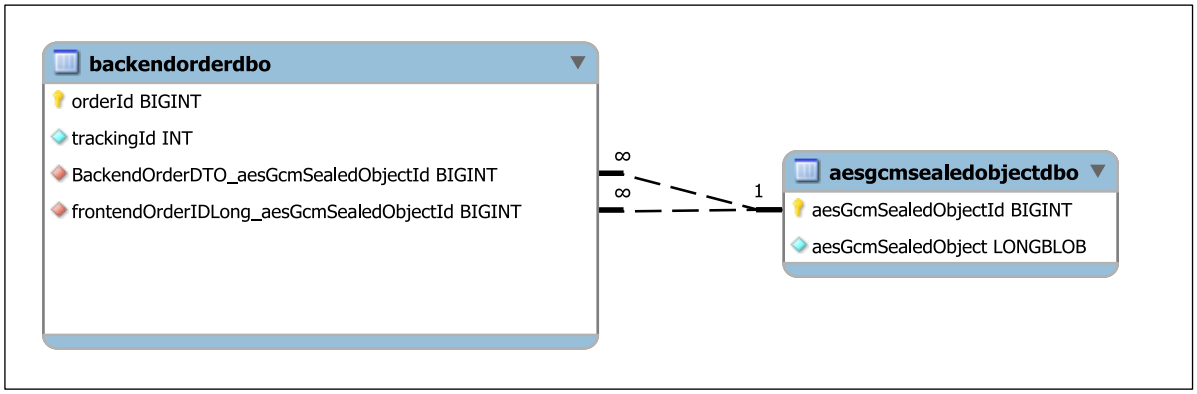

Figure 39: Backend Application database schema

Appendix O

In-Page Sorting on Shop Page

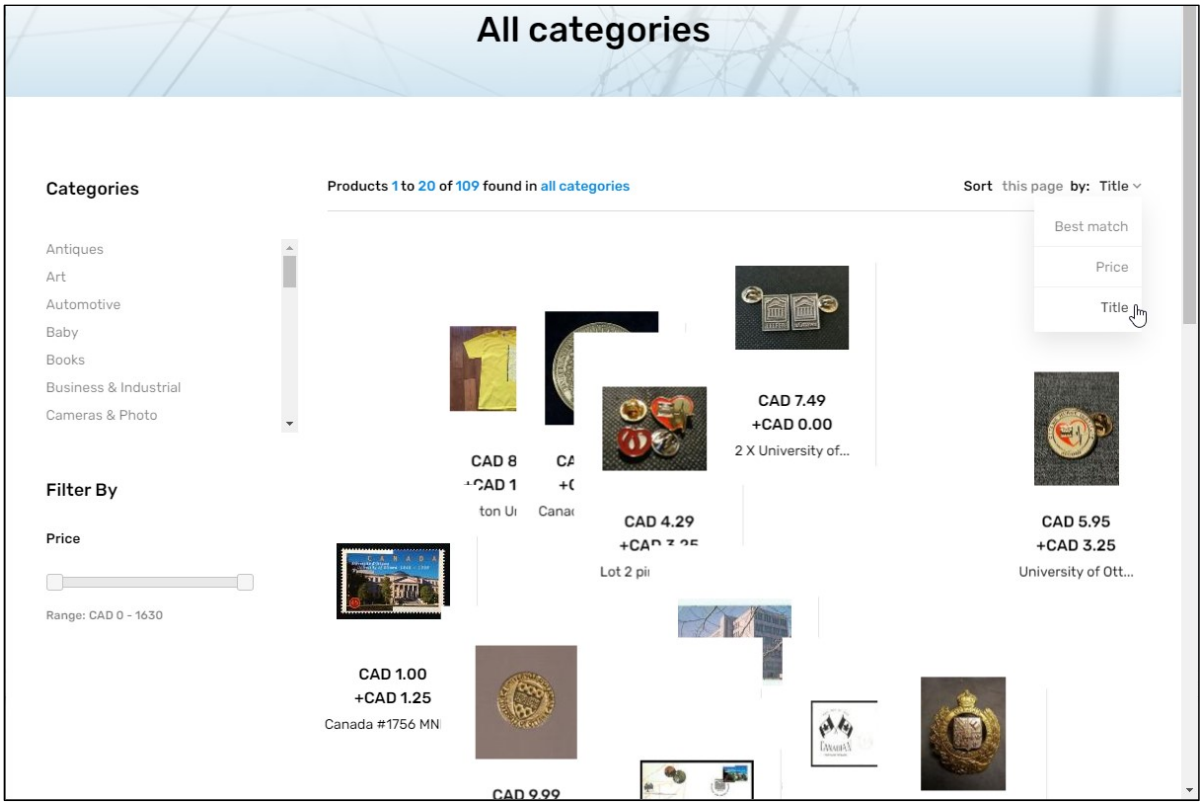

Figure 40: In-page sorting on Shop Page

Appendix P - Cryptographic Algorithm Implementations used in GhostBuy Prototype

Table 14: Cryptographic algorithm implementations used in GhostBuy Prototype

Application	Component	Implemented in	parameters, e.g. Algorithm, Mode, Padding	Security Level	based on	Source
Portal Application, Frontend Application, Backend Application	Portal Controllers (parent class = GhostBuyPortalController), Web Services (parent classes = GhostBuyFrontendRestfulWebService, GhostBuyBackendRestfulWebService), DAO Implementations (parent classes = GhostBuyBackendDAOImplementation, GhostBuyFrontendDAOImplementation)	AesGcmCrypto (custom)	AES/GCM/NoPadding with 12 Byte Initialization Vector and 128 Bit Tag and usage of associated data	256 Bit AES	Java crypto functions from javax.crypto and java.security.SecureRandom with "Windows-PRNG")	n/a
Backend Application	Web Service (GhostBuyBackendRestfulWebService)	RsaOaepCrypto (custom)	RSA/ECB/OAEPPADDING (ECB effectively means "none") with SHA-512 (Message Digest) and MGF1 (Mask Generation Function) and SHA-512 (Message Digest of Mask Generation Function)	4096 Bit (modulus), about 149 Bit AES equivalency	Java crypto functions from javax.crypto and java.security.SecureRandom with "Windows-PRNG")	n/a
Portal Application, Backend Application	Portal Controllers (CheckOutController, GetCredentialHashingParametersController. SigninController, SignupController), Backend Web Services (ShoppingCartProcessService)	PBKDF2Crypto (custom)	PBKDF2 algorithm ("PBKDF2WithHmacSHA512") with 100.000 iterations (client password rehashing on server), 1 iteration (order cart hashing) - each with secret random 256 bit salt (stored with AES256-encryption)	256 Bit Collision Resistance	Java javax.crypto.SecretKeyFactory	n/a
Client Application	ghostbuy_module	aes_gcm_functions (custom)	AES/GCM/ n/a with 12 Byte Initialization Vector and 128 Bit Tag and usage of associated data	256 Bit AES	WebCrypo API: window.crypto.subtle.generateKey, .importKey, .exportKey, .encrypt, .decrypt	n/a
Client Application	ghostbuy_module	rsa_oaep_functions (custom)	RSA / n/a / OAEP with SHA-512 (Hashing Algorithm = Message Digest), other parameters presumably as for custom Java class RsaOaepCrypto since only used for decryption	4096 Bit (modulus), about 149 Bit AES equivalency	WebCrypo API: window.crypto.subtle.generateKey, .importKey, .exportKey, .encrypt, .decrypt	n/a
Client Application	ghostbuy_module	general_crypto_functions (custom)	PBKDF2 algorithm (name:"PBKDF2", hash: "SHA-512") with 10.000 iterations (client password hashing), 100.000 iterations (password based client key derivation), 1 iteration (order cart hashing) - each with secret random 256 bit salt (stored and transmitted AES256 encrypted)	256 Bit Collision Resistance (only 256 bit used for password-based key)	WebCrypo API: window.crypto.subtle.deriveBits	n/a
Client Application	ghostbuy_module	general_crypto_functions (custom)	SHA-256 with secret random 256 bit salt (stored with AES256-encryption)	256 Bit Collision Resistance	WebCrypo API: crypto.subtle.digest	n/a
(Portal Application), Frontend Application, Backend Application	FrontendWebServiceClient, BackendWebServiceClient, consuming web services that run on Tomcat v9.0 Server	JAXRSClientUtil (custom)	Server configuration: TLSv1.2 and ciphers="TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" define: ECDHE for key exchange and afterwards AES/GCM (NoPadding) with 128 Bit Tag	256 Bit AES	Java javax.ws.rs.client.Client	https://tools.ietf.org/html/rfc5288
(Client Application), Portal Application	Browser (Google Chrome, New Microsoft Edge, Mozilla Firefox, Opera) and Tomcat v9.0 web server	https implementation of browser running Client Application / browsing Portal Application served by Tomcat v9.0 web server	Server configuration: TLSv1.2 and ciphers="TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" define: ECDHE for key exchange and afterwards AES/GCM (NoPadding) with 128 Bit Tag	256 Bit AES		https://tools.ietf.org/html/rfc5288
Frontend Application, Backend Application	Frontend & Backend DAO Implementations	HibernateConnectionUtil (implemented 2x, each for Frontend and Backend), class was adapted almost 1:1 from bookshop project	TLS/SSL Connection with "ECDHE-RSA-AES256-GCM-SHA384"	256 Bit AES	uses JDBC conncetion configured in respective hibernate.cfg xml, (SQL Server configured with self-signed certificate)	n/a
(Client Application), Portal Application	Session Cookie in Browser (Google Chrome, New Microsoft Edge, Mozilla Firefox, Opera) and Tomcat v9.0 web server	Client Application / browsing Portal Application served by Tomcat v9.0 web server Session Manager App	<Manager secureRandomAlgorithm="Windows-PRNG"> <SessionIdGenerator sessionIdLength="32"/> </Manager>	256 Bit Session ID	Tomcat v9.0 web server implementation (SessionIdGeneratorBase.java)	

Appendix Q

Storage Consumption of IndexedDB

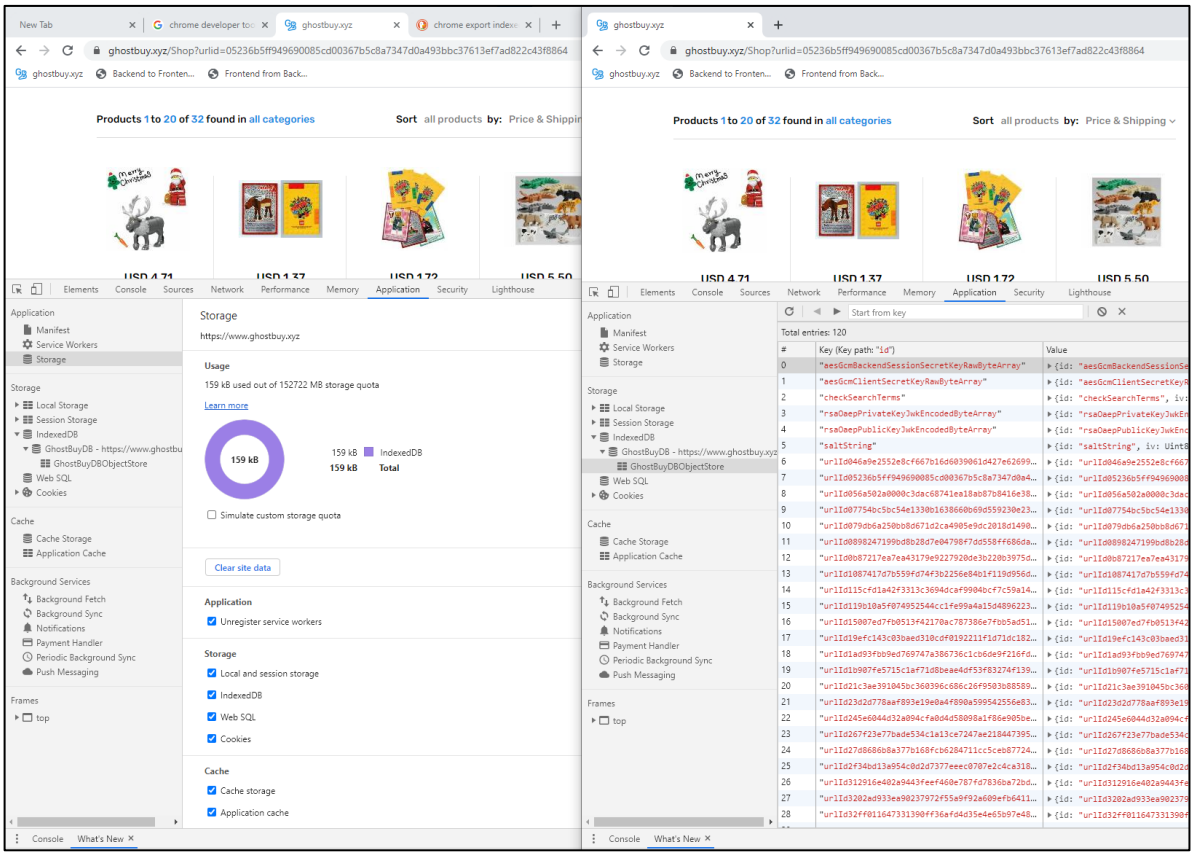

Figure 41: Storage consumption of IndexedDB with 114 different pages visited

Appendix R - Performance Measurements of GhostBuy Prototype on Laptop PC

Table 15: Performance measurements of GhostBuy Prototype on laptop PC

#	Browsing Flow Step	Description	Run	Products retrieved in this step			*Client Application* (Browser)													*Backend Application*			
							Resources			Time Consumption in ms										Approximate Time Consumption in			
														Scripting									
				Retrieved #	Image size after scaling	Information & image source	Bytes loaded	Bytes transferred	Bytes / # of images	Total	Time / # of Images	System (e.g. profiling)	Loading (e.g. parse html, CSS)	Microtasks (execution)	Other (e.g. compile)	Rendering	Painting	Key Generation + Decryption (if appl.)	Idle (waiting)	Total	Query online shop (eBay)	Image retrieval & scaling	Other (e.g. logging, processing, encryption)
1	Initial Cryptographic Setup	Open portal URL (→ key generation & key exchange, shopping cart creation, home page load)	1st	n/a	n/a	n/a	D 6,375,698	2,105,288	n/a	**5,477**	n/a	328	132	137	464	104	13	3,423 B + 12	864	28	n/a	n/a	28
2	Home Page Reload	Click logo on home page (→ home page reload)	(1st)	n/a	n/a	n/a																	
3	Product Search	Search for "EBT" in shop "eBay Canada"	1st	20	small	eBay																	
4	Home Page Load & Shop Preselection	Click logo link in shop page header (→ home page load, preselects shop)	(1st)	n/a	n/a	n/a	Steps performed but data of client not recorded																
5	Product Search	Search for "EBT" in shop "eBay Canada"	2nd of #3	20	small	eBay																	
6	Product Search	Search for "EBT" in shop "eBay Canada" by selecting "100 products/page"	(1st)	E 100	small	eBay	4,822,554	E 1,924,275	19,243	**5,512**	55	226	95	758	523	128	36		3,746	2760	694	1903	163
7	Product Search	Search again for "EBT" in shop "eBay Canada" by retriggering search in shop page header	Similar to #6	100	small	eBay	4,774,916	1,918,257	19,183	**3,852**	39	286	153	566	132	75	15		2,625	1919	537	1273	109
8	Product Details	Open product details of "HOn30 East Broad Top 23 1/2 Foot Early Flatcar Kit 2 Pack EBT"	1st	10	large	eBay																	
9	Add to Shopping Cart	Add product "HOn30 East..." to shopping cart	1st	1	med.	eBay																	
10	Product Details	Open product details of "HOn30 East ..." (by clicking image in shopping cart)	2nd of #8	10	large	eBay																	
11	Add to Shopping Cart	Add product "HOn30 East..." to shopping cart	2nd of #9	1	med.	eBay																	
12	Decrease Item Quantity	Decrease item quantity of product "HOn30 East..." in shopping cart (1x = down to 1)	1st	1	med.	cart	Steps performed but data of client not recorded																
12-2	Increase Item Quantity	Increase item quantity of product "HOn30 East..." in shopping cart (1x = up to 2)	(1st)	1	med.	cart																	
12-3	Decrease Item Quantity	Decrease item quantity of product "HOn30 East..." in shopping cart (1x = down to 1)	2nd of #12	1	med.	cart																	
13	Start checkout	Click "Checkout" (redirects to sign-in page)	(1st)	n/a	n/a	n/a																	
14	Sign-In	Perform Sign-In (click "sign in" after entering test account data)	(1st)	1	med.	cart	2,475,285	155,766	155,766	**2,556**	2,556	106	24	100	182	62	13	C 73	1,996	processed by Portal & Frontend Application			
15	Continue Checkout	Click "Continue Checkout" (after entering shipping and payment information)	(1st)	1	med.	cart	Steps performed but data of client not recorded																
16	Place Order	Click "Place your order"	(1st)	n/a	n/a	n/a																	
17	Initial Cryptographic Setup	Open portal URL (→ key generation & key exchange, shopping cart creation, home page load)	2nd of #1	n/a	n/a	n/a	D 6,464,127	56,332	n/a	**3,161**	n/a	233	62	108	271	96	11	1,880 B + 11	489	11	n/a	n/a	11

Appendix S - Less significant or more obvious limitations of the GhostBuy Prototype

Table 16: Less significant or more obvious limitations of the GhostBuy Prototype

#	Category	Limitation	Description	Potential Measures to Overcome the Limitation
1	GUI / Workflow	Suboptimal QR Code sticker	The QR code sticker that is used for "transmitting" the encrypted remote order IDs during the package handover is rather big and potentially recordable.	The QR code could be replaced by an RFID (nearfield?) sticker, which cannot be easily recorded by employees. If the QR code should still be used, its size could be reduced by optimizing the encrypted content. E.g. the content could be a leaner AES256 ciphertext as opposed to the sealed object that we use currently.
2	GUI	Mobile Device Experience	The *Client Application* does not display well on mobile devices (with small screen sizes).	More effort could be put into optimizing the *Client Application* GUI for mobile devices, e.g. through more thorough modifications and testing of the "responsive" CSS files.
3	GUI	Usability issues	Test users of our implementation did not grasp the GUI concept right away.	A real-world implementation should seek to use UI design research to create a better, easier to understand GUI (which for example requires less reading).
4	GUI	Wordlist for password generator to be cleaned	The wordlist DB that we use for password generation was initially checked by us to remove certain offensive terms but it still contains words which should be removed.	A more thorough deletion of somehow offensive words should be done regarding the wordlist for the password generator.
5	Security	Limited wordlist for password generator	We found that the wordlist that we used for the password generator was big enough for the prototype implementation. However, the effective password strength is limited to about 61 bits, which is good but could be further improved.	More secure passwords could be generated by using larger word corpuses and/or allowing for more words to be used in the passphrase.
6	Security	Protection against database parameter (SQL) injection	The prototype implementation barely sanitizes search terms and other input parameters on the server-side. This could create a potential attack vector for adversaries.	We assume that the usage of the Hibernate ORM mitigates the risk of SQL injections happening in the platform's databases; this should be verified. In addition, search terms should be more thoroughly sanitized before being forwarded to the eBay API.

Appendix S

#	Category	Limitation	Description	Potential Measures to Overcome the Limitation
7	Security	No check of crypto parameters that are generated on the server side.	We implemented the prototype in a way that most encryption and hashing parameters are generated on the server side to avoid that predefined / unsafe parameters are used by clients (e.g. due to manipulated *Client Application* script files). However, we do not perform a server-side check if these parameters have actually been correctly adapted but instead just save the parameters returned by the client. Obviously, this does not bar adversaries from manipulating the parameters.	The server-side generated crypto parameters should be checked to match with those parameters that are returned by the client. This would require at least a brief caching of the parameters (other than being subsequently saved to the database).
8	Security	No client authentication by webservice	We consider the *GhostBuy Platform* to be a save environment. However, the fact that potentially any client could contact the platforms webservices would be a security risk in a real-world environment (even though they often need to provide secret knowledge to obtain data).	Web service clients should authenticate themselves before being granted access to the platform's web services. This could be achieved by using client certificates or authentication tokens.
9	Security / GUI	No timeout on sign-up and sign-in page	The sign-up and sign-in pages could potentially sit idle for a long time while someone has already entered their credentials. These could be obtained by adversaries.	The sign-in and sign-up pages should feature timers that either clear the input forms or at least mask the most sensitive input fields (passwords).
10	Performance	Sequential request-handling of server applications	The *Portal Application's* controllers and the *Frontend Application* and *Backend Application's* web services handle requests sequentially/synchronously which would not perform under heavy workloads.	The *Portal Application's* controllers and the *Frontend Application* and *Backend Application's* web services should be implemented in a way that allows for asynchronous request handling.
11	Performance	Sequential code processing in *Client Application*	Although many parts of the *Client Application's* processing use parallelized concepts, e.g. such as concurrent AJAX requests, many potentially parallelizable code structures still exist.	The *Client Application's* source code (especially ghostbuy_module.js) could be examined for parallelizable code structures – in some cases we added according comments. Those code structures could in most cases be executed parallelized by using the "Promise.all" functionality.
12	Source Code	Too specific crypto classes	The function / methods that we use throughout the implementation often accept specific encrypted parameters and are even named accordingly. This makes switching to other encryption algorithms more difficult.	Using encryption-algorithm specific parameters and function / method names should be avoided. Instead, more generic names should be used and more generic parameters should be used for input and output types.

Appendix T

Example of Tamper-Proof Tape and Seal

Figure 42: Tamper-proof tape[107]

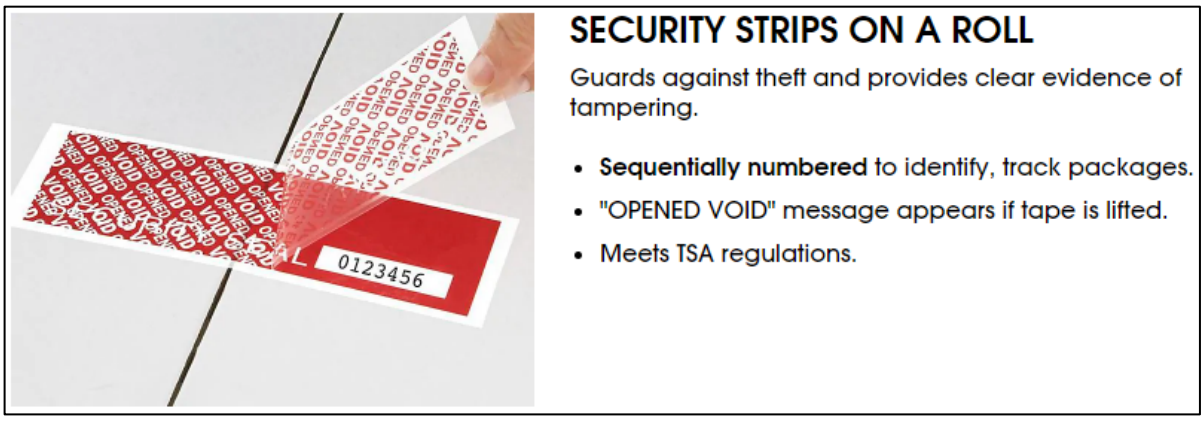

Figure 43: Tamper-proof seal / security strip[108]

[107] Source: https://www.uline.ca/BL_520/Hidden-Message-Security-Tape

[108] Source: https://www.uline.ca/BL_3093/Security-Strips-on-a-Roll

Appendix U

Research Problems and Solutions/Ideas

Table 17: Research problems and solutions/ideas

#	Short Problem Description	Solution / Idea
1)	The *Frontend Company* and the *Backend Company* each have to be able to refer to the *Customer* without any employee being able to collaborate "offline", i.e. separate from the platform`s secure environment, to match/link datasets about the Customer stored in each company's database.	The link between the records in the database is made by placing encrypted order IDs in each others database (see Section **5.6.9**)
2	Maliciously collaborating employees should be detectable.	The platform would need to keep a record of which employee processed which order, respectively which employee might have obtained/recorded which customer related data (see Section 6.3.6). This includes that any access to data is being logged.
3	Customer information transmitted to the *Backend Company* or vice versa, must not be obtained by the *Frontend Company*. However, the type of information being exchanged must be controllable in order to avoid certain types of personal identifiable information being exchanged between them. E.g. the Backend Company may not obtain the name or IP address of C.	The customer and the Backend Company communicate via AES256-encrypted information exchange after the *Initial Cryptographic Setup* has been build. The communication is controlled/forwarded by the *Portal Application* and *Client Application (see Section* **5.5***ff.)*
4	The platform must have a means of communicating with the Customer regarding any previous transaction even after the direct interaction – in which this transaction took place – with the platform ended (e.g. the Customer closed the browser and the web session ended).	The respective *Backend Session Key (6-B/C)* is encrypted with a password key based key that is generated by the *Client Application (see Section* 5.6.9*).*
5	The platform must be able to provide data for criminal investigations in a privacy preserving fashion.	See Section **5.10**.

Appendix U

#	Short Problem Description	Solution / Idea
6	Cold start problem in databases (too few customers make initial customers easy to identify / link to orders).	Add random entries beforehand. Do not provide interfaces for cleartext insight (in bulk). Encrypt database (see sections 5.7, 5.8).
7	Packages received by *Backend Company* have to be linked to respective order.	Tracking ID is specified as part of shipping address (e.g. in C/O). Tracking ID can be non-numerical if necessary (see Section **5.7**)
8	Packages received by *Backend Company* have to be documented	Delivery note is scanned and stored with according record (encrypted).
9	Correlation attacks from banks, web shops, shippers are possible.	See Section **6.2.1**)
10	Someone could try to identify content e.g. by known weight of products.	Weights will likely not reveal the exact product (because weight might be too similar, several products might be contained etc.)
11	Shipper could use other means (than opening) to identify content of package (x-ray, weight).	In general one should assume that the necessary effort to do so is too high (and it is illegal). Also x-rays might not reveal what actually is in the package (additionally, the package could be lined with metal foil (aluminum) to make it harder to x-ray). X-rays might show same picture for different products (e.g. not show what exact DVD was ordered). Weight: See problem 10.
12	Shipper opens packages before delivery to customer.	Use a package seal and have the customer verify the delivery in unopened condition. See Section 5.10.
13	*Bank 1, Shipper 2 and Frontend Company* know customer information and that customer bought from *Frontend Company.*	Significance of problem is reduced since customers cannot be linked to specific or a certain type of product if the offered choice of products is big. Therefore, the platform integrate at least one online retailer / marketplace with a large variety of products such as Amazon, Walmart, BestBuy and eBay.
14	Similar to 6: If only few orders are processed per day employees can more easily collaborate e.g. with online shop to find correlations.	Significance of problem is reduced as compared to not using the platform. Customer is within anonymity set and their size can be influenced. Also the effort for collaborating will be too high for common online shops.

Appendix U

#	Short Problem Description	Solution / Idea
15	This kind of architecture might not allow for anonymous purchasing of prescription drugs.	Might be approached as like anonymous election (outer (digital) envelope contains verification that receipt is approved, inner (digital) envelope contains receipt). Another solution might be to use digital credentials.
16	Do *Frontend Company* and *Backend Company* trust each other? What if they do not?	Use hash values to verify transactions (see Section 5.6.9). Blockchain might be used to improve on that. Some exploiting the system (e.g. overcharging customer) can be identified by monitoring the system / auditing the implementation.
17	Architecture creates additional costs and uses more resources (monetary, time, energy, waste…)	That is the price for anonymity, but certain measures can help reduce these costs (see Section **6.5.1**).
18	A platform like this potentially could be used by criminals.	Platform only allows for purchasing legal goods. Money laundry (probably) as (un-)likely as on other platforms since criminal investigations are possible.
19	Minors might get products that they are not eligible to receive.	Either allow only adults to use platform or request proof of age at *Frontend Company* (always, or only if *Backend Company* asks for it) and/or implement data transfer (via OT) on being minor/adult between *Frontend Company* and *Backend Company*: Backend only actually asks for age proof status if specific goods shall be ordered.

Appendix V

Sources that were particularly helpful for the implementation

1. Web Crypto API:
    - Overview: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API
    - Implementation examples: https://github.com/diafygi/webcrypto-examples
    - Live table to test browser's support of Web Crypto API: https://diafygi.github.io/webcrypto-examples/
2. Web Page Template:
    - Library: https://colorlib.com/wp/templates/
    - OneTech – the template used by us: https://colorlib.com/wp/template/onetech/
3. Generation of animated logos:
    - https://icons8.com/preloaders/en/letters_numbers_words
4. Jquery UI (e.g. interactive input masks):
    - https://jqueryui.com/ (1.12.1 used)
5. Font (e.g. review star icon):
    - https://fontawesome.com/ (fontawesome-free-5.13.1-web used)
6. Extended JavaScript regular expressions:
    - http://xregexp.com/
    - https://travis-ci.org/github/slevithan/xregexp, respectively
    - https://github.com/slevithan/xregexp (4.3.0 used)
7. Java libraries that we imported via Maven:
    - Logging:
        - https://mvnrepository.com/artifact/org.slf4j/slf4j-api (1.7.30 used)
        - https://mvnrepository.com/artifact/org.slf4j/slf4j-log4j12 (1.7.30 used)
        - https://mvnrepository.com/artifact/log4j/log4j (1.2.17 used)
    - Java object ↔ JSON conversion: https://mvnrepository.com/artifact/com.google.code.gson/gson (2.8.6 used)
    - Base class for web service client:
        - https://mvnrepository.com/artifact/org.glassfish.jersey.bundles/jaxrs-ri

(1.30 used, with <type>zip</type> and <classifier>src</classifier>)

- Database implementation:
  - https://mvnrepository.com/artifact/mysql/mysql-connector-java (8.0.21 used)
  - https://mvnrepository.com/artifact/org.hibernate/hibernate-core (5.4.20.Final used)
  - https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager (5.4.20.Final used)
- QR code generation:
  - https://mvnrepository.com/artifact/com.google.zxing/core (3.4.0 used)
  - https://mvnrepository.com/artifact/com.google.zxing/javase (3.4.0 used)
- eBay API authentication:
  - https://mvnrepository.com/artifact/com.ebay.auth/ebay-oauth-java-client
  - (1.1.0 used)
- String manipulation (e.g. for html unescaping)
  - https://mvnrepository.com/artifact/org.apache.commons/commons-text (1.8 used)

Appendix W

Accompanying Files

We provide the files that accompany this thesis in a webpage that can be accessed via this link: http://thesis.ghostbuy.xyz/

As of now, the link redirects to a OneDrive folder.

The OneDrive folder contains the following subfolders and files:

1. Referenced unpublished document
    - 2019-02-25 Literature Review - Anonymous Online Shopping - Fabian Willems.pdf
2. Demo videos
    - 1 Before Watching - Please Readme.txt
    - 2 GhostBuy Masterthesis - Overview Demo.mp4
    - 3 GhostBuy Masterthesis - Detailed Demo.mp4
3. Source code, documentation & implementation description
    - Projects.zip
    - Documentation.zip (JavaDoc and JSDoc)
    - Implementation Description.pdf
4. Performance analysis data
    - Desktop Performance.zip
    - Laptop Performance.zip