Refactoring Misuse Case Diagrams using Model Transformation

Mohamed El-Attar and Nidal Nasser
Department of Software Engineering, College of Engineering, Alfaisal University, Riyadh, Saudi Arabia

Keywords: Misuse Case Modeling, Refactoring, Model Transformation, Antipatterns.

Abstract: Secure software engineering entails that security concerns needs to be considered from the early phases of development, as early as the requirements engineering phase. Misuse cases is a well-known security analysis and specifications techniques, based on the popular use case modeling technique, that takes place in the requirements engineering phase. Similar to use case modeling, misuse case modellers are prone to committing modeling mistakes and applying antipatterns. As a result, misuse case models need to be analysed to determine if they contain fallacious design decisions. Changes, known as refactoring, to the misuse case diagrams are then required to remedy any design issues and such changes which would normally be manually applied. However, manual application of such changes in misuse case models are prone to human error, further compounding the design issues in a given misuse case model. To this end, this paper presents a model transformation approach to systematically apply changes to misuse case models. A case study related to a book store is presented to illustrate the application and feasibility of the approach.

1 INTRODUCTION

Security is an essential quality attribute that needs to be considered during the requirements engineering phase. Use case modeling (Booch et al., 2005; Jacobson, 1992; Bittner and Spence, 2002; OMG, 2011) is already a very popular technique to elicit, analyse and model functional requirements of a system. Misuse case modeling (Sindre and Opdahl, 2005) is a technique, based on use case modeling, that can be used to elicit, analyse and specify security requirements. A misuse case describes expected system behaviour similarly to a use case, except they describe negative operational sequences that can lead to a system being compromised.

Similar to use case modeling, a misuse case model can be improperly designed containing errors that may lead to critical security threats not being addressed. Therefore, it is critical to create high quality misuse case models. Certain problematic designs aspects can be repeatedly committed when modellers create their misuse case models, referred to as antipatterns. One approach to remedy fallacious designs in misuse case models is to detect these structural antipatterns and refactor them. Refactoring a misuse case model will alter its structural design to eliminate potential problems resulting from the original design and resetting the misuse case models to properly specify security requirements in the way security modellers have intended.

In earlier work, an approach to assess and improve the quality of misuse case diagrams based on detecting antipatterns and apply refactorings was presented (El-Attar, 2012). However, the approach presented in (El-Attar, 2012) does not provide tool support to automatically detect antipatterns and apply the refactorings, entailing a manual application. Detecting antipatterns and applying modeling refactorings is far from straightforward process. Hence, a manual application of the approach presented in (El-Attar, 2012) can be prone to many human errors, further compounding the design issues in a faulty misuse case diagram. To this end, this paper presented a model transformation approach to detect antipatterns in misuse case diagrams and to apply its corresponding refactorings, preventing potential human errors.

The remainder of this paper is organized as follows: Section 2 provides the necessary background related to the misuse case diagram notation. Section 2 also describes the approach of improving misuse case diagrams by detecting structural antipatterns and applying their corresponding refactoring. In Section 3, an approach to detect antipatterns in misuse case diagrams and applying their refactorings using model transformation is described. A case study is

El-Attar, M. and Nasser, N.
Refactoring Misuse Case Diagrams using Model Transformation.
DOI: 10.5220/0007675502490256
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 249-256
ISBN: 978-989-758-375-9

presented in Section 4 to demonstrate the application and feasibility of the proposed model transformation approach. Finally, Section 5 concludes and suggests future work.

2 REFACTORING MISUSE CASE MODELS BASED ON ANTIPATTERNS

This section provides necessary background on the misuse case modeling notation and the use of antipatterns to drive misuse case diagram refactorings.

2.1 Misuse Case Diagrams Notation

Misuse case diagrams subsumes the entire notational set of use case diagrams (Sindre and Opdahl, 2005). A misuse case describes harmful behaviour in the form of step-based scenarios, similar to how use cases describe business related functional behaviour. A misuse case is depicted as a black oval, to signify its inverse relationship with use cases. Misuse cases can share original relationships such as the *include, extend* and *generalization* relationships. In addition, misuse case diagrams introduce the concept of a misuser. A misuser is analogous to an actor. A misuse case can only be associated with misuse cases similar to how actor can only be associated with a use case. A misuser is an external entity that accessed a misuse case to execute behaviour that can compromise a system. A misuse case can access the misuse case with malicious intentions or unintentionally. A misuser is depicted as a black stickman figure, to signify its inverse relationship with an actor. Finally, misuse case diagrams introduce the concepts of the *threatens* and *mitigates* relationship. A *threatens* relationship can only be directed from a misuse case to a use case to indicate that the harmful behaviour contain in the misuse case can be executed to negatively affect the business related behaviour contained in the use case. A *mitigates* relationship can only be directed from a use case to a misuse case to indicate that the security related behaviour in the given use case can be executed to mitigate the threated posed be the given misuse case.

2.2 Refactoring Misuse Case Diagrams based on Antipatterns

An antipattern is defined as a "literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences". In the context of software engineering requirement-oriented and design-oriented diagrams, an antipattern will describe an unsound structure and its potential harmful consequences downstream in the development process. An antipattern also provides key information on how it can be detected within a diagram and how the diagram should be altered to remedy the faulty design. Altering a diagram is performed by applying one or more refactorings.

It should be noted that a detected antipattern does not definitively prove the existence of a detect. An antipattern detection will only prompt modellers to re-evaluate their design. While assessing their diagrams due to an antipattern detection, the modellers will reference key information provided in the antipattern description itself to determine if an error indeed exists, prompting the application of one or more refactorings.

An approach to improve the quality use case diagrams based on antipatterns was presented in (El-Attar and Miller, 2006, 2010, 2012; Khan and El-Attar, 2016). The approach was then extended to improve the quality of misuse case diagrams in (El-Attar, 2012). A summary of the taxonomy of antipatterns presented in (El-Attar, 2012) is shown below in Table 1. Table 1 also presents the corresponding refactorings for each antipattern. Due to space restrictions, only antipatterns "a1." and "a2." only will discussed in more details.

A1. Accessing a Generalized Concrete Misuse Case

This antipattern is concerned with the case that a misuser is associated directed with a *generalized* misuse case in order to enable one of its *specializing* misuse cases. A *generalized* misuse case would normally contain abstract and incomplete behaviour that is common amongst all of its *specializing* misuse cases. However, a direct association with a *concrete generalized* misuse case may result in executing this *generalized* misuse case without executing any of its *specializing* misuse cases resulting in incomplete meaningless behaviour being executed.

This antipattern can be remedied by applying one of two refactorings. Refactoring **"r1."** entails the *concrete generalized* misuse case to be set as an *abstract* misuse case, forcing the execution of one of its *specializing* misuse cases. The second refactoring **"r2."** requires that the direct association between the misuser an the *concrete generalized* misuse case to be dropped and replaced with direct associations from the misuser to the misuse case's *specializing* misuse cases, once again, forcing the execution of one of the *specializing* misuse cases.

Figure 1 presents an instance of antipattern **"a1."**. Figures 2 and 3 present that application of refactorings **"r1."** And **"r2."**. respectively.

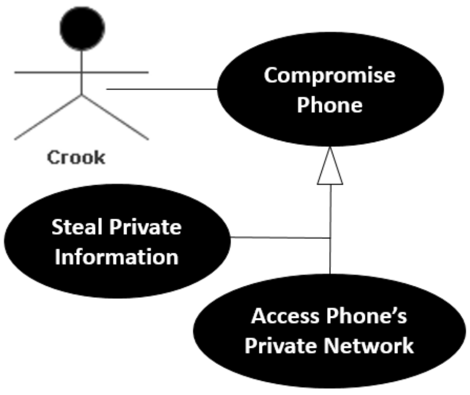

Figure 1: Antipattern **"a1."** Example.

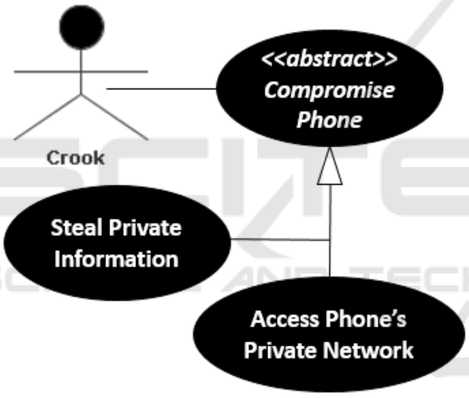
Crook
<<abstract>> Compromise Phone
Steal Private Information
Access Phone's Private Network

Figure 2: Applying refactoring **"r1."**.

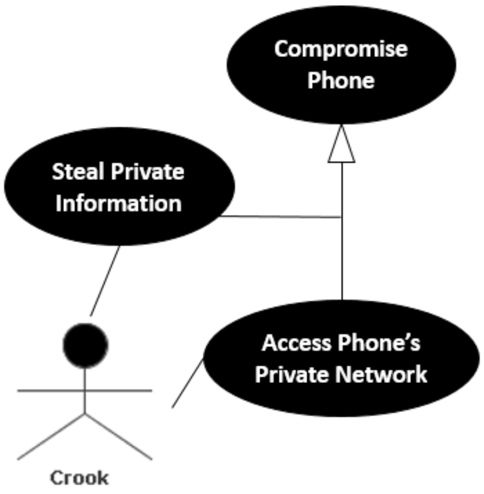

Figure 3: Applying refactoring **"r2."**.

A2. Accessing an Extension Misuse Case

This antipattern is concerned with the case that a misuser is associated directly with an *extension* misuse case. A direct associated between a misuser and a misuse case can lead to the execution of the *extension* misuse case only. This is inappropriate design since *extension* misuse cases are specifically tailored to be an extension of its corresponding *base* misuse case. Hence, an *extension* misuse case would be rather meaningless if executed independently from its *base* misuse case. This design applied by modellers allow a misuser to provide information to an executing *extension* misuse case. However, it is the *base* misuse case that should receive information required by the *extension* misuse case from the misuser.

Antipattern **"a3."** can be remedied by applying either refactoring **"r3."** or **"r4."**. Refactoring **"r3."** requires the direct association between the misuser and the *extension* misuse case to be removed, hence preventing the possibility of the misuser executing the *extension* misuse case independently from its *base* misuse case. Meanwhile, refactoring **"r4."** requires the association relationship between the misuser and the *extension* misuse case to be directed from the *extension* misuse case to the misuser. A direction association only allows the *extension* misuse case to initiate the engagement with the misuser, given that the *base* misuse case is properly executed, while preventing the misuser from initiating the engagement with the *extension* misuse case whereby there is a potential that the *extension* misuse case may be executed independently from its *base* misuse case.

Figure 4 presents an instance of antipattern **"a2."**. Figures 5 and 6 present that application of refactorings **"r3."** And **"r4."**. respectively.

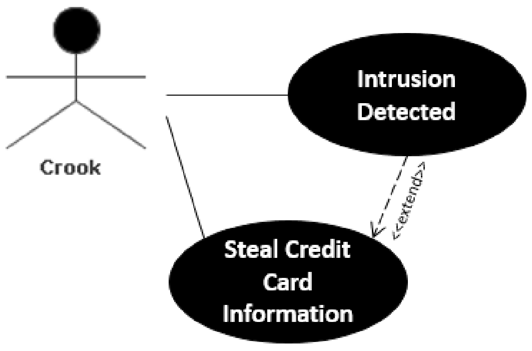

Figure 4: Antipattern **"a2."** Example.

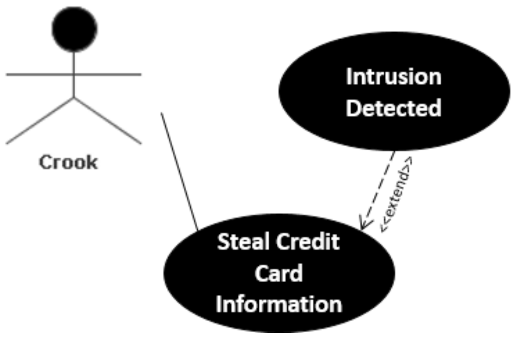

Figure 5: Applying refactoring **"r3."**.

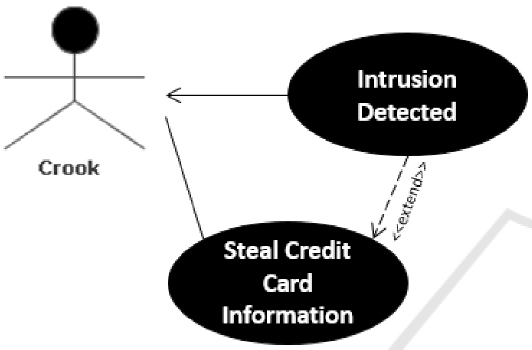

Figure 6: Applying refactoring **"r4."**.

3 REFACTORING MISUSE CASE DIAGAMS USING MODEL TRANSFORMATION

A model transformation is an automated conversion of a source model to a target model based on a set of predefined transformation rules. A rule described the mapping of source model elements to target model elements. A rule also describes the conditions which triggers the transformation of a source model (or a sub-part of the source model) to become a target model.

A metamodel for misuse case diagrams need to be defined as a prerequisite to model transformations. The target and source models are the same (misuse case diagrams) and hence they are represented using the same metamodel, hence an endogenous model transformation is performed. The metamodel developed utilizes many elements from the official UML specifications document [OMG]

There exist many model transformation languages that can be used and implemented with various strengths and weaknesses. The model transformation language used in this paper Is ATL (ATL Transformation Language) (ATLAS Group, 2006). ATL provides a very beneficial advantage in that it provides declarative and imperative programming capabilities to implement transformations. The transformation implemented in this research work using a combination of both programming paradigms. Transformation algorithms are defined as a set of ATL modules which are comprised of a set of ATL rules and helpers. ATL rules and helpers define how a target model instance is generated from a source model instance. Due to space restrictions, the remainder of this section presents the ATL modules that were developed to

Table 1: Misuse case antipatterns and their respective refactorings.

Antipattern	Refactoring
a1. Accessing a *generalized concrete* misuse case	r1. Concrete to Abstract r2. Drop Misuser-Generalized MUC Association
a2. Accessing an *extension* misuse case	r3. Drop Misuser - Extension MUC Association r4. Directed Misuser - Extension MUC Association
a3. Using *extension*/*inclusion* misuse cases to implement an *abstract* misuse case	r5. Abstract Extended MUC to Concrete r6. Inclusion to Generalization
a4. Functional Decomposition: Using the *include* relationship	r7. Drop Functional Decomposition r8. Drop Functional Decomposition having Inclusion
a5. Functional Decomposition: Using the *extend* relationship	r9. Split Extension MUC r10. Extension to Generalization
a6. Multiple *generalizations* of a misuse case	r11. Generalization to Include
a7. Misuse cases containing common and exceptional functionality	r12. Drop Inclusion r13. Drop Extension
a8. Multiple misusers associated with one misuse case	r14. Generalize Misusers r15. Split MUCs
a9. An association between two misusers	r16. Drop Misuser-Misuser Association
a10. An association between misuse cases	r17. Drop MUC-MUC Association
a11. An unassociated misuse case	r18. Drop Unassociated UC
a12. Two misusers with same name	r19. Rename Misuser
a13. An misuser associated with an unimplemented *abstract* misuse case	r20. Abstract to Concrete r21. Add Concrete MUC

```
module MUC_AP_2_R1;
create umlP: UML refining umlAp: UML;

--helper for accessing the misuse
helper def: actor : UML!Misuser =
   UML!Misuser ->allInstances()->first();

--helper for accessing the specialized misuse
cases
helper def: specializedMUCs :
Sequence(UML!MisuseCase)
   = UML!MisuseCase-> allInstances()->select
       (a|a.generalization->size()>0);

--helper for accessing the first specialized
misuse cases
helper def: firstSpecializedMUC : UML!MisuseCase
   = thisModule.specializedMUCs->first();

--helper for accessing the second specialized
misuse cases
helper def: secondSpecializedMUC :
UML!MisuseCase
   = thisModule.specializedMUCs->last();

--Declarative matched rule for refining package
rule Package_To_Package {
   from s: UML!Package in umlAp
   to t: UML!Package in umlP (
     packagedElement<-Sequence
   {s.packagedElement , a2}
   ),
   a2: UML!Association in umlP (
     memberEnd <- a2p1,
     navigableOwnedEnd <- a2p2,
     ownedEnd <- Sequence{a2p1, a2p2}
   ),
   a2p1: UML!Property in umlP (
     name <- 'src',
     association <- a2,
     type <- thisModule.misuser
   )
   a2p2: UML!Property in umlP (
     name <- 'dst',
     association <- a2,
     type <- thisModule.secondSpecializedMUC
   )
}

--Declarative matched rule for refining
Assocations
rule Association_To_Assocation {
   from a: UML!Association in umlAp
   to a1: UML!Association in umlP (
     name <- a.name,
     ownedEnd <- a.ownedEnd
   )
}

--Declarative matched rule for refining
Association Properties
rule Property_To_Property {
   from s: UML!Property in umlAp
   to t: UML!Property in umlP (
    name <- s.name,
    type <- if s.name = 'src'
     then thisModule.misuser
       else
     if s.name = 'dst'
     then thisModule.firstSpecializedMUC
     else
             s.debug('Error')
     endif
       endif
   )
}
```

Listing 1: The ATL code to apply the refactoring using explicit associations with specializing misuse cases.

implement the refactoring of antipattern “**a1.**” And “**a2.**”, discussed previously in Section 2.

As discussed previously in Section 2, antipattern “**a1.**” requires that the association between the misuser and the generalized misuse case cases to be replaced with associations between the misuser and the specializing misuse case cases. Listing 1 presents the ATL code which applies this refactoring. The transformation commences by executing the rule `Package_To_Package` as the *package* element is considered to be the root node. The target diagram is named `umlP` while the source diagram is named `umlAp`. The `Association_To_Association` rule is then invoked to perform the actual refactoring. Invocation of this rules changes the association end from the generalized misuse case to one of the specializing misuse cases. Association model elements are then created connecting the misuser with the specializing misuse cases.

A second refactoring is subsequently executed. The second refactoring is based on setting the generalized misuse case as `abstract`. The second refactoring is implemented using the `MisuseCase_To_MisuseCase` rule. The rule simply sets the `isAbstract` attribute of the misuse case element to `true` (see Listing 2).

```
module MUC_AP_2_R2;
create umlP: UML refining umlAp: UML;

rule MisuseCase_To_MisuseCase {
     from s: UML! MisuseCase in umlAp (
          s.generalization->size() = 0
     )
     to t: UML! MisuseCase in umlP (
          isAbstract <- true
     )
}
```

Listing 2: The ATL code to apply the refactoring by setting the parent misuse case as *abstract.*

```
rule DropMisuseCase {
  from s: UML!MisuseCase  (
    not(s.isAssociated() or s.isIncluded() or
s.isIncluder() or s.isExtended()
    or s.isGeneralization() or
s.isSpecialization()) and s.extend->size()>1;
  )
  to drop
  do {
    for(ex in s.extend) {

thisModule.AddBaseMisuseCaseforExtension(ex);
    }
  }
}
```

Listing 3: *Implementation* of the first refactoring for antipatterns “a1.”

Antipattern "**a2.**" relates to the improper usage of the *extend* relationship as discussed previously in Section 2. This antipatterns instance is remedied by applying two refactorings. The first refactoring is implemented by the rule `DropMisuseCase` in Listing 3. Rule `DropMisuseCase` checks for *extension* misuse cases that are shared by multiple *base* misuse cases, then proceeds to delete them when found. An invocation to `AddMisuseCase` adds specific *extension* misuse cases into the misuse case model for each of the *base* misuse cases. The name of the newly generated misuse cases is appended with the name of its corresponding *base* misuse case.

The second refactoring required to remedy the "**a2.**" antipatterns instance is implemented by the rule `DropExtend` shown in Listing 4. Rule `DropExtend` check for *extend* relationships who *extension* misuse case is shared with multiple *base* misuse cases and then proceeds to delete them when found. A call to `AddGeneralization` results in generating a *generalization* relationship from the *extension* misuse case to its corresponding *base* misuse case.

```
rule DropExtend {
  from s: UML!Extend   (
    not (s.getExtension().isAssociated() or
s.getExtension().isIncluded() or
    s.getExtension().isIncluder() or
s.getExtension().isExtended() or
    s.getExtension().isGeneralization() or
s.getExtension().isSpecialization()) and
    s.getExtension().extend->size() > 1

  )
  to drop
  do {
    thisModule.AddGeneralization(s);
  }
}
```

Listing 4: ATL rule for applying Extension to Generalization refactoring.

4 ONLINE STORE CASE STUDY

The Online Store (OS) system allows its customers to submit online orders and to send and receive email from store personnel. The security threats that were identified in the requirements phase include stealing a customer's credit card information, spreading malicious code and email interceptions. To mitigate against these threats, the system is supplemented with an encryption based defensive mechanism as countermeasure. The misuse case diagram developed is shown in Figure 7.

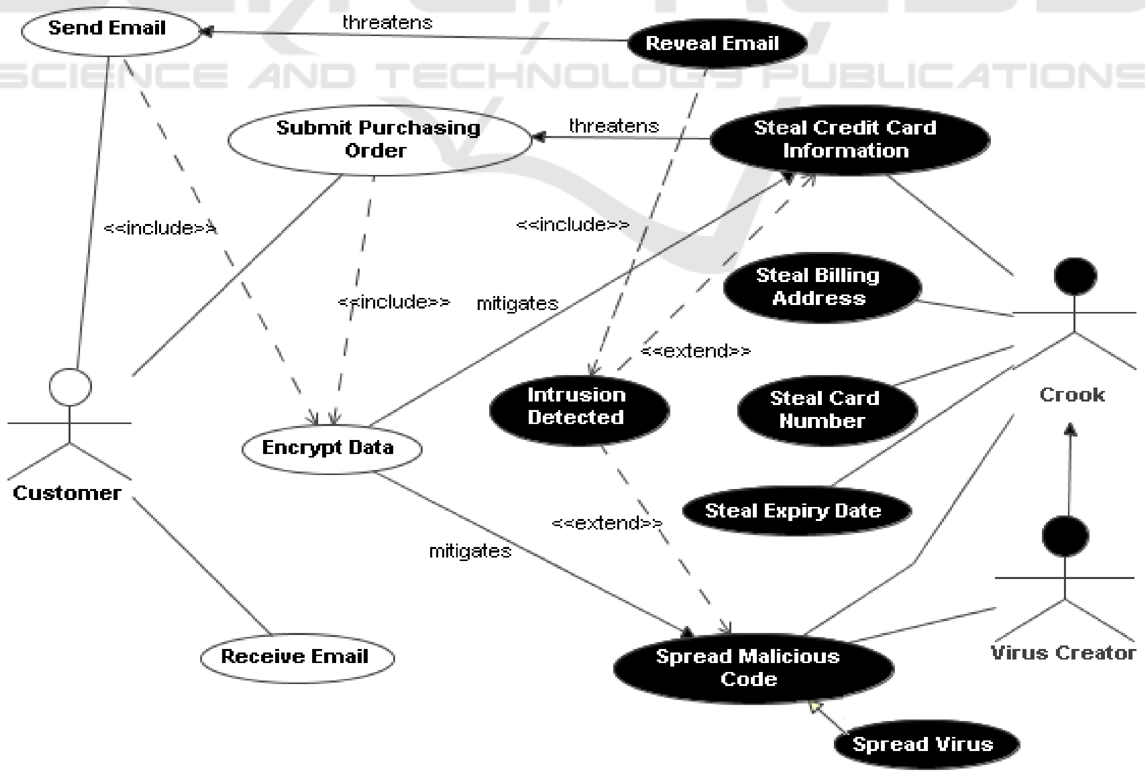

Figure 7: The original misuse case diagram of OBS.

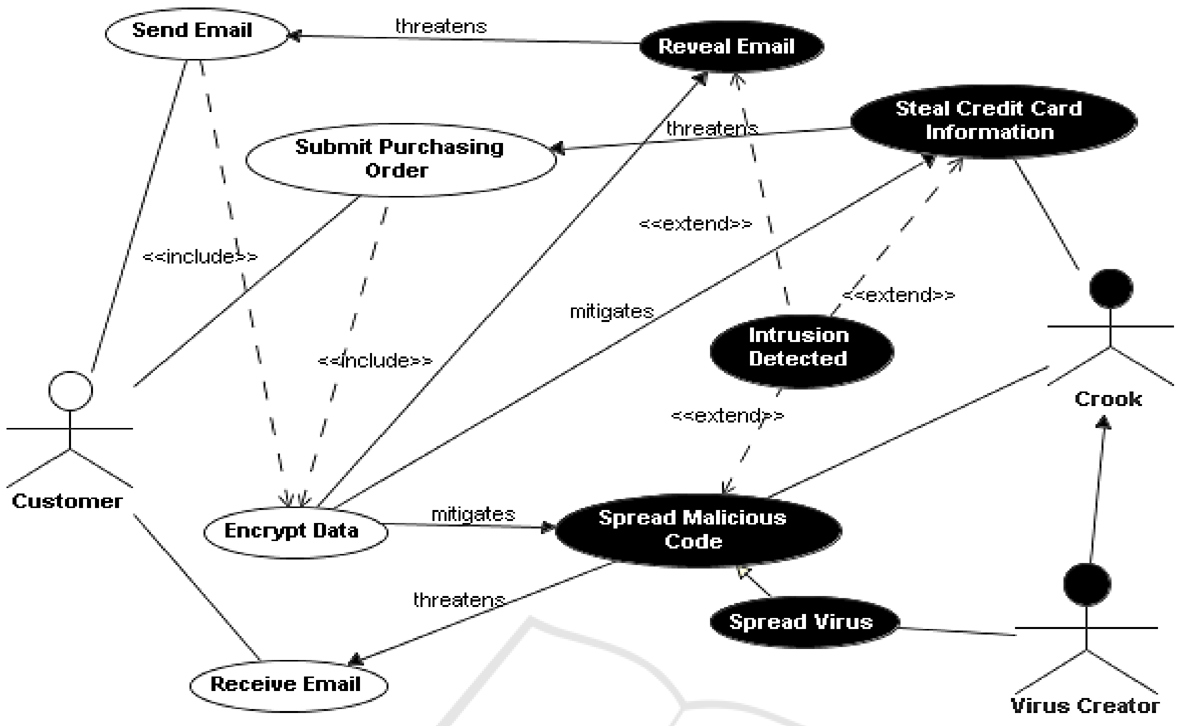

Figure 8: The misuse case diagram of OBS after applying the refactorings in response to addressing the antipattern matches.

In its current state, the OS misuse case model suffers from a number of problems hampering its quality and reducing the comprehension of the security threats. Table 2 below presents the antipatterns that were detected in the OBS misuse case model. A total of 8 antipatterns instances were detected.
The proposed model transformation approach automatically detected these antipatterns and applied its corresponding refactorings. The results are reflected in the misuse case diagram shown in Figure 8.

Table 2: OS misuse case model antipattern matches.

Match No.	Antipattern Matched	Elements involved
1	**a8.** Functional decomposition: using pre and postconditions	Misuse Cases: Steal Credit Card Information, Steal Billing Address, Steal Card Number and Steal Expiry Date
2	**a5.** Functional decomposition: using the *include* relationship	Misuse Cases: Reveal Email and Intrusion Detected
3	**a5.** Functional decomposition: using the *extend* relationship	Misuse Cases: Intrusion Detected, Steal Credit Card Information, Spread Malicious Code
4	**a5.** A non-threatening base misuse case	Misuse Cases: Steal Billing Address, Steal Card Number, Spread Malicious Code and Spread Virus
5	**a5.** Multiple misusers associated with one misuse case	Misuse Cases: Spread Malicious Code Misusers: Crook and Virus Creator
6	**a4.** A misuse case that is not associated with any misusers	Misuse Cases: Spread Virus
7	**a6.** An unmitigated base misuse case	Misuse Cases: Spread Virus, Reveal Email, Steal Billing Address, Steal Card Number and Steal Expiry Date
8	**a6.** A described misuser that is not depicted in the diagram	Misuser: Get Admin Privileges

5 CONCLUSIONS AND FUTURE WORK

Misuse case modeling is one of the more commonly known security modeling techniques that can be deployed at the requirements engineering phase. The quality of misuse case diagrams produced significantly affects the security addressing efforts of a development team. Many misuse case diagrams contain design defects that can mislead the development team. A taxonomy of these defects has been developed in previous work entailing the detection of structural antipatterns and a number of refactorings to be applied based on detected antipatterns. This paper proposes a model transformation approach to detect antipatterns and apply refactorings to eliminate the potential of human error when performing these two main activities manually. The proposed approach was applied to a generic online store case study that demonstrated its application and feasibility.

Future work can be directed towards created extending the exiting taxonomy of misuse case modeling and to provide model transformation-based implementations of their associated antipattern detection and refactoring activities. Other future work can be directed towards utilizing model transformation to refactoring other requirements and design phase models.

REFERENCES

ATLAS Group, 2006. ATL User Manual. [online] Available at:. < http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf> [Accessed: 1 February 2012]

Bittner, K. and Spence, I., Use Case Modeling. Addison-Wesley, 2002.

Booch, G., Rumbaugh, J., and Jacobson, I., The Unified Modeling Language User Guide, Second Edition. Addison-Wesley, 2005.

El-Attar, M. and Miller, J., "Matching antipatterns to improve the quality of use case models," in Requirements Engineering, 14th IEEE International Conference, 2006, pp. 99-108.

El-Attar, M. and Miller, J., “Improving the Quality of Use Case Models Using Antipatterns,” J. of Software and Systems Modeling, 2010.

El-Attar, M. “A Framework for Improving Quality in Misuse Case Models.” Business Proc. Manag. Journal 18(2): 168-196 (2012).

El-Attar, M. and Miller, J., "Constructing high quality use case models: a systematic review of current practices," Requirements Engineering, vol. 17, pp. 187-201, 2012.

Jacobson, I. et al.: Object-Oriented Software Engineering. A Use Case Driven Approach. Addison-Wesley (1992)

Khan, Y. and El-Attar, M., “Using Model Transformation to Refactor Use Case Models Based on Antipatterns”. Information Systems Frontiers 18(1): 171-204 (2016)

Object Management Group (OMG), 2011. OMG Unified Modeling Language (OMG UML) Superstructure. <https://www.omg.org/spec/UML/2.4.1/About-UML/> [Accessed: 19 October 2018]

Sindre, G., Opdahl, A., Eliciting security requirements with misuse cases, Requirements Engineering Journal, vol 10, pp. 34-44 (2005)