Integrating Formal Analysis and Design to Preserve Security Properties

Riham Hassan  
Department of Computer Science, Virginia Tech  
rhabdel@cs.vt.edu

Shawn Bohner  
Department of Computer Science, Virginia Tech  
sbohner@vt.edu

Sherif El-Kassas  
Department of Computer Science, The American University in Cairo  
sherif@aucegypt.edu

Michael Hinchey  
Rose-Hulman Institute of Technology, University of Limerick  
mike.hinchey@lero.ie

Abstract

The use of formal methods has long been advocated in the development of secure systems. Yet, methods for deriving design from requirements that guarantee retention of the intended security properties remain largely unrealized on a repeatable and consistent basis. We present the FADES (Formal Analysis and Design approach for Engineering Security) that integrates KAOS (Knowledge Acquisition in autOmated Specifications) with the B specification language to derive security design specifications and further implementation from security requirements. We demonstrate the capability of the approach to handle changes to security requirements by introducing corrective changes to the security requirements of a case study, the spy network system. The objective is to bridge the gap between formal requirements and design for security requirements. Our initial results show promise with FADES in preserving security properties and detecting security vulnerabilities early during requirements. Encouraged by these, we are more quantitatively assessing the FADES capabilities.

1. Introduction

Security is a growing concern as we develop increasingly large and complex systems supporting more distributed and integrated capabilities in the public and private sectors [1]. According to the CERT Coordination Center (CERT/CC), security incidents reported have grown exponentially over the past decade [3]. The impacts on society are still being felt.

According to Viega and McGraw, one of the major sources of vulnerability has been poor-quality software [9]. Security is usually considered after the fact and often in an ad-hoc manner resulting in insecure software products. Even in security evaluation, securing software is recognized as a function of formal specification and transformation -- as evidenced with the requisite formalism in Common Criteria Evaluation Assessment Levels (EAL) 5, 6 and 7.

Security requirements play a crucial role in designing secure software. With poor specification of security requirements, many security issues become vulnerabilities that are pushed to maintenance for resolution [2]. Yet, in practice, it has not been clear how to engineer security systematically throughout the software lifecycle [1].

Lack of rigor in specifying and designing security requirements directly contributes to the problem of poor software security. A formal security requirements elaboration method encapsulated in an uncomplicated interface and transformed to a formal language is needed to rigorously derive design specifications that maintain properties of the requirements model.

The use of formal methods for producing highly assured software has long been accepted and advised for developing secure systems [14]. For security requirements, the extra effort of employing formal methods in specifying security can be justified when rigor applies to security specific elements of software product (allocating the cost to the vulnerable aspects of the software). However, in practice, the extra cost is traded off against alleviating substantial losses that may result from poor security engineering practices.

The ineffective handling of changes to software security specifications often results in the introduction of vulnerabilities. Conflicts with existing security features along with the lack of procedural resolution strategies are significant reasons behind poorly secured software products. Software security evolution still lacks clear and consistent procedures that deal with security specifications changes in a provable way. Further, insufficient traceability information at key stages of software development makes it difficult to handle changes in security specifications.
In this paper, we address some of these concerns through the FADES approach. We outline the method and present a case study of the security requirements for a spy network system that has been devised by Fontaine in [7]. The spy network system broadcasts secret revelations into a network of spies around the world. We model and implement the security requirements of the spy network system using FADES to preserve security properties. Further, we illustrate the mechanisms embedded in FADES to efficiently handle changes of security requirements and analyze security vulnerabilities resulting from change.

Following this introduction, section 2 outlines relevant related work and the underlying technologies employed by FADES; namely KAOS security extension and B. Section 3 describes the FADES approach and how it handles security-specific elements. Section 4 develops security requirements of the spy network system using FADES. Section 5 gives an example of corrective changes to security requirements and how to implement them using FADES. Section 6 outlines our preliminary conclusion.

2. Background

This section outlines research efforts in employing formal methods to software security and briefly describes the two underlying technologies employed in FADES, which are KAOS security extension described in [1, 4] and the B formal development method. KAOS was selected as it presents a systematic approach to capturing the essence of security requirements in a convenient and understandable form (a goal graph). B was selected based on its effective mechanisms for systematic elaboration and refinement and good tool base.

2.1 Related work

Stepney et. al. in [15] have presented a case study of an electronic purses system that carry financial value, each hosted on a smart card. Stepney et. al. in [15] have used Z as a formal development platform to construct a security requirements specifications model that has been further refined to derive design and implementation specifications. Our initial results of comparing FADES to Stepney et. al. approach have shown that FADES is more capable in building a complete and consistent security requirements model with the ability to perform threat analysis and mitigation as it employs KAOS for requirements analysis while Stepney’s approach specifies the system directly in Z.

Massacci, et. al. in [10] have presented a comprehensive case study of the application of secure TROPOS requirements engineering method for compliance to the Italian legislation on privacy and data protection. Secure TROPOS is close in spirit to KAOS except that TROPOS starts with agents rather than goals. Unlike our work in this paper that has been able to derive design and implementation from the security requirements of the SNS system, Massacci et. al. work has only led to the definition and analysis of ISO-17799-like security management scheme [10].

2.2 Extending KAOS for security

Van Lamsweerde in [4] has described a general approach for eliciting, analyzing and modeling functional and non-functional requirements of software systems based on KAOS framework that employs first-order temporal logic. He then customized his goal-oriented approach to handle security requirements in specific [1]. This subsection focuses on the security specific approach.

KAOS is based on considering requirements as goals to be achieved by the envisioned system. A goal is, “a prescriptive statement of intent about some systems whose satisfaction requires the cooperation of some of the system agents while agents are active components such as humans, devices, legacy software or software-to-be components” [1]. Van Lamsweerde differentiates between goals and domain properties (i.e., descriptive statements about the environment such as physical laws, organization norms or policies) [1].

Goals are organized in a hierarchy obtained from the refinement of higher level goals into lower-level goals using the AND/OR refinement mechanism. Higher-level goals are strategic, coarse-grained and involve multiple agents while lower-level goals are technical, fine-grained and involve fewer agents [1]. The resulting hierarchy is a directed, acyclic-graph. AND-refinement relates a goal to a set of subgoals where the satisfaction of the higher-level goal requires the satisfaction of all its subgoals. The OR-refinement relates a goal to a set of alternative subgoals in which the achievement of the higher-level goal requires the achievement of at least on of its subgoals.

Goal refinement ends when every subgoal is realizable by some individual agent assigned to it [1]. A requirement is a terminal goal in the goal graph assigned to an agent (active object) in the software-to-be while an assumption is a terminal goal assigned to an agent in the environment. In the security requirements context, assumptions might be used to capture security policies in the environment. Goals are “operationialized” into operations performed by agents to achieve these goals. Goals and operations refer to
entities (passive objects) that are incrementally derived from the goals specifications to produce a structural model of the system [1].

Obstacles are undesirable goals that hinder desired goals from being achieved. Obstacles are elaborated as goals and Van Lamseweerde in [4] has defined a formal systematic procedure for identifying all the possible obstacles of a certain goal. Obstacles are then analyzed based on their criticality to the well-being of the whole system and resolution mechanisms are used to generate alternative resolutions such as goal substitution and obstacle prevention.

In the security requirements context, attackers are malicious agents in the environment and threats are obstacles intentionally set up by attackers while assets are to be protected against threats [1]. Van Lamsweerde has specified some generic specification patterns for eliciting security goals such as confidentiality, integrity, availability, privacy, authentication and non-repudiation. The general method for elaborating security requirements is to systematically iterate the following steps:

- Instantiate specification patterns associated with property classes.
- Derive anti-model specifications threatening such specifications.
- Derive alternative countermeasures and define new requirements or modify some existing requirements.

2.3 The B Method

B is a method for specifying, designing and coding software systems that integrates formal proof techniques in software development [11, 16]. The basic idea of B development is to start with an abstract model of the system under development while more detailed models are built by refining the initial model. A key merit of this refinement mechanism is the ability to preserve already proven system properties in higher level models. B models are accompanied by mathematical proofs, called proof obligations that guarantee correctness and effectiveness of system development.

The B method is based on the Abstract Machine Notation (AMN), which provides structuring mechanisms to support modularity and abstraction in an object-based style [16] as each B machine encapsulates its variables and operations. At the most abstract level it is obligatory to describe the static properties of a model’s data by means of an “invariant” predicate [11]. This gives rise to proof obligations used to guarantee development correctness [11]. Security engineering is a good domain to make use of the correctness of development capabilities of the B method. Software systems with security requirements exhibit a strong need for assured development that allows for adding more details on an abstract system model without violating the already proven security properties.

B models rarely need to make assumptions about the size of the system being developed [12]. The price to pay is to face possibly complex mathematical theories and difficult proofs; however, the re-use of developed models and the structuring mechanisms available in B help decrease the complexity [11].

3. The FADES Approach

This section summarizes the FADES approach that is being applied to the spy network system in order to model and implement its security requirements. A more detailed treatise is provided in [8].

FADES is a requirements-driven software engineering approach that derives design specifications from a set of security requirements modeled using KAOS security extension framework. FADES provides a secure software engineering methodology that integrates the KAOS security extension [1], which is characterized by the ability to formally build a complete, consistent and clear requirements model with the B method, which provides formal guarantees for the correctness of the system development. Our research showed that KAOS is promising in that it could be extended with an extra step of formality in order to fully implement security requirements while preserving the requirements properties of completeness, consistency and clarity [8]. Moreover, extending KAOS with more formality in a development framework like B allows for tracing requirements at the various steps of development; that is, during both the design and implementation.

FADES elaborates security requirements using KAOS in order to build a complete, consistent and unambiguous requirements model. The KAOS security extension is based on modeling security requirements as goals, which is relatively intuitive and close to the way people perceive requirements as expressed in natural language. Further, the method formally analyzes possible threats and performs threat mitigation while building the security requirements model resulting in the detection and resolution of security vulnerabilities very early in development.

The choice of B as a formal development platform for elaborating security design specifications assists in preserving security properties of requirements when design specifications are being derived. B has the notion of model refinement that allows for building a
detailed model of design from an abstract model of requirements while preserving the security properties of the requirements model.

Employing formal methods in FADES provides a reasonable approach to the challenge of developing secure software products with formal evidence of correctness. Recognizing that formal methods reduce security risks but entail more cost, we justify this cost by applying FADES only to security, which is a critical aspect of the system. Further, software systems that are evaluated for security at the Common Criteria (CC) EAL 6 and 7 need formal evidence assuring the security of the software. This makes software products developed using the FADES approach securely compliant with CC higher levels.

FADES provides means for transforming the security requirements model built with KAOS to an equivalent one in B using some transformation rules that are briefly described. The whole system is represented either as a single abstract B machine or multiple abstract B machines related to each other based on the size of the system. Each KAOS entity (passive object) is represented as a B machine encapsulating its KAOS attributes, representing the system state, in the form of the B machine variables and the KAOS domain properties in the form of the B machine invariant. The system machine(s) include entity machines. Each KAOS operation is represented as a B operation in the system machine and uses the entities either as parameters or return values. The precondition of each KAOS operation is directly mapped to the precondition of the corresponding B operation since both are represented in first-order predicate logic. Post conditions of KAOS operations have no direct mapping in B and should be enforced by the B operation specification and refinement. KAOS goals are not transformed to B as they are indirectly transformed through KAOS operations that realize these goals.

The B model that has been transformed from KAOS representing security requirements is then refined using non-trivial B refinements that generate design specifications conforming to the security requirements. Figure 1 illustrates a general model of the FADES approach. FADES allows for deriving one artifact, which is design from another artifact, which is a requirements model using formal representation. Hence, the links between artifacts are clear enough to provide traceability information.

Traceability information is available in FADES for both forward (tracing requirements to design) and backward (tracing design decisions to requirements) traceability. Forward traceability from requirements to design could be shown through the refinement steps in B that builds a more detailed model from a more abstract one. This means that FADES is capable of building traceability links from each requirement to its realization in design through the refinement steps in B. Backward traceability of design specifications to the original set of security requirements can be formally shown through the B proof obligations that show how each B refinement step preserves the constraints of the abstract model being refined.

4. The Spy Network System (SNS)

This section examines the FADES approach through a case study of a spy network system designed by Fontaine [7]. Unlike many case studies in the security literature, the SNS case study is of a size; small enough to be manageable and large enough to be convincing. Further, the SNS has a set of security requirements that are commonly mandated in most communication-based systems making it a good framework for applying FADES to this category of systems. The spy network case study has been derived from two real case studies, the British National Health Service (NHS) and the eBay online auction website. The next subsections briefly describe the SNS security requirements followed by discussing how to elaborate these requirements in KAOS. We then outline how the KAOS requirements model is transformed to B followed by refining the B model to obtain security design specifications and generate implementation.

4.1 The SNS security requirements
Fontaine in [7] has described the SNS that aims at broadcasting secret revelations into a network of spies around the world. The big boss supervises all missions, allocates spies to missions, and appoints bosses to teams. Spies collect revelations about the enemy and target them to other members of the team working on the mission. The SNS has the following security requirements:

- **Revelation integrity**: every copy of a revelation should be identical to the original revelation written by the author.
- **Revelation confidentiality**: a revelation might only be known by spies working in the mission.
- **Revelation authentication**: every revelation is attributable to an author and the purported author of the revelation should be correct.
- **Mailbox access control**: a mailbox should only be accessed by its subscribed spy.

### 4.2 KAOS Security Requirements Model

Fontaine has applied the KAOS security extension to the spy network case study [7]. We have used Objectiver (a commercial software requirements modeling tool built by Van Lamsweerde and his team who have created the KAOS framework) as a tool to build the KAOS requirements model. The high level goal of the SNS is to make the system secure defined as:

**Goal Maintain [RevelationIntegrity]**

**InformalDef** A copy of a revelation is identical to the original

**InstanceOf** IntegrityGoal, AccuracyGoal

**FormalDef** ∀ sp1, sp2 : Spy, re1, re2 : Revelation

Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧ CopyOf(re1, re2) ⇒ re1.Content = re2.Content

**Goal Maintain [RevelationConfidentiality]**

**InformalDef** The revelation may be known only by spies working in the mission which the revelation is about

**InstanceOf** ConfidentialityGoal

**FormalDef** ∀ sp, re : Spy, revelation, re1, re2 : Team

Member(sp, re1) ∧ Targeted(re, re2) ∧ re1 ≠ re2 ∧ ¬Knows(sp, re1.Content)

**Goal Maintain [RevelationAuthentication]**

**InformalDef** Agents can verify who has authorship on a revelation

**InstanceOf** AuthenticationGoal

**FormalDef** ∀ sp1, sp2 : Spy, re, re : Revelation

AuthorOf(sp1, re) ∧ Owning(sp2, re) ⇒ Knows(sp2, AuthorOf(sp1, re))

**Goal Maintain [MailboxAccessControl]**

**InformalDef** Mailboxes are accessed only by their subscribed spy

**InstanceOf** AccessControlGoal

**FormalDef** ∀ sp : Spy, ma : Mailbox

Accessed(ma, sp) ⇒ Subscribed(ma, sp)

The second step to build the security requirements goal graph is to analyze and refine the security goals elaborated in the first step by finding conflicts and obstacles to security, that is, perform threat analysis and resolution. Due to space limitations, we will show how to find threats to the RevelationIntegrity goal as an example. Resolution strategies to potential security threats are proposed.

In order to obtain the security threats to the goal RevelationIntegrity, we start by negating the goal RevelationIntegrity to get the following:

**Obstacle RecipientHasCorruptedRevelation**

**InformalDef** Recipient owns a corrupted copy of the revelation

**InstanceOf** AccessControlGoal

**FormalDef** ∃ sp1, sp2, re1, re2 : Revelation

Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧ CopyOf(re1, re2) ∧ re1.Content = re2.Content

The following object model increment is required

**Relationship Corrupted**

**InformalDef** A revelation copy is not intact with respect to the original revelation

**Links** Revelation (Role Characterized, Card 0..N)

**DomainVar**

∀ re1, re2 : Revelation

Corrupted(re1) ∨ Corrupted(re2) ⇒ CopyOf(re1, re2) ∧ re1.Content = re2.Content

∀ sp1, sp2, sp3 : Spy, re1, re2 : Revelation, me1, me2 : Message

Receiving(sp2, me1, sp1) ∧ Sending(sp2, me, sp3) ∧ About(me1, re1)

∧ About(me2, re2) ∧ CopyOf(re1, re2) ∧ re1.Content = re2.Content ⇒ Corrupted(re2)

∀ sp, re : Revelation

Collecting(sp, re) ⇒ ¬Corrupted(re)

Regressing through the Corrupted relationship, we get

**Obstacle RecipientHasCorruptedRevelation**

**InformalDef** An agent has sent a different revelation than the one he has received

**InstanceOf** AccessControlGoal

**FormalDef** ∃ sp1, sp2, re1, re2 : Revelation

Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧

[Receiving(sp2, me1, sp1) ∧ Sending(sp2, me, sp3) ∧ About(me1, re1)]

∧ About(me2, re2) ∧ CopyOf(re1, re2) ∧ re1.Content = re2.Content ⇒ Corrupted(re1)

A strong mitigation would be achieved through the following goal:

**Goal Achieve [WhoTeaminformedWhenCorrupted]**
This goal is further refined in multiple steps that are not shown here due to space limitation.

The third step in building the security goal graph is to resolve obstacles. We show the resolution of the obstacle to the goal RevelationIntegrity as follows: a way to operationalize the RevelationIntegrity goal is to have digital signatures. Each revelation has a signature that depends on the revelation text and the author's identity.

After building the goal graph, leaf goals are assigned to agents responsible for achieving these goals through means of operations. KAOS operations are used to operationalize (i.e. fulfill) requirements. Operations constrain the space of solutions that could be used by solution providers to design the system that shall meet the requirements. The step of goal operationalization bridges the gap between the problem description space (requirements) and the solution space (design) [4]. The following operations have been obtained during goal operationalization.

**Operation SignRevelation**

Input: Spy{arg sp}, Revelation{arg rev}, Privatekey {arg pk}

Output: Signature {res sig}

DomPre → (rev1:Revelation) Signed(rev1, sig, pk)

DomPost (rev2:Revelation) Signed(rev2, sig, pk)

ReqPreFor RevelationSignedWhenSent

**Operation VerifyRevelation**

Input: Spy{arg sp}, Revelation{arg rev}, Publickey {arg pk}, Signature {arg sig}

Output: Boolean {res verified}

DomPre → (rev1:Revelation) Verified(rev1, sig, pk)

DomPost (rev2:Revelation) Verified(rev2, sig, pk)

ReqPreFor RevelationVerifiedWhenReceived

ReqPreFor RevelationDecryptedWhenReceived

**Operation EncryptRevelation**

Input: Spy{arg sp}, Revelation{arg rev}, Publickey {arg pk}

Output: Message {res msg}

DomPre → (rev1:Revelation) Encrypted(rev1, pk)

DomPost (rev2:Revelation) Encrypted(rev2, pk)

ReqPreFor RevelationEncryptedWhenSent

ReqPostFor RevelationEncryptedWhenSent

**Operation DecryptRevelation**

Input: Spy{arg sp}, Message{arg msg}, Privatekey {arg pk}

Output: Revelation {res rev}

DomPre → (msg1:Message) Decrypted(msg1, pk)

DomPost (msg2:Message) Decrypted(msg2, pk)

ReqPreFor RevelationDecryptedWhenReceived

ReqPreFor RevelationDecryptedWhenReceived

**Operation CertifySpy**

Input: Spy{arg sp}, Publickey{arg pk}

Output: Boolean {res Authenticated}

DomPre → (sp:Spy) Certified(sp, pk)

DomPost (sp:Spy) Certified(sp, pk)

ReqPreFor RevelationVerifiedWhenReceived

ReqPostFor RevelationEncryptedWhenSent

**Operation AccessMailbox**

Input: Spy{arg sp}, Mailbox{arg ma}, Password {arg pa}

Output: Boolean {res accessed}

DomPre → (ma:Mailbox) Accessed(ma, sp)

DomPost (ma:Mailbox) Accessed(ma, sp)

ReqPreFor MailboxAccessControl

ReqPreFor MailboxAccessWithPassword

The above operations are transformed to B to construct the initial B machine that is further refined.
inside B using the B refinement mechanism to derive design specifications and generate implementation. The rational for transforming KAOS operations to B while not transforming the rest of the goal graph is that operations sums up all the behaviors that agents need to have to fulfill their requirements, which are the leaf goals in the goal graph. The mechanism of constructing the goal graph shows that high level goals are refined using AND/OR refinement steps until leaf goals are derived meaning that the fulfillment of leaf goals implies the fulfillment of the higher level goals in the goal graph. Therefore, it is safe to only transform KAOS operations used to express behaviors of agents that perform them to fulfill the leaf goals in the goal graph without compromising the completeness and consistency properties of the requirements model.

4.3 Transforming Security Requirements to B

The initial B machine obtained from transforming the requirements model obtained in section 3.2 to B includes an abstract representation of each KAOS operation as a B operation. KAOS operations preconditions are mapped to preconditions of their corresponding B operations. Machine invariants in B are constructed from the invariants of the KAOS objects manipulated by KAOS goals. For further details on the transformation scheme, refer to [8]. We have employed the B-Toolkit, which is one of the two most famous commercial tools for B development as a tool to develop our B model and refine it to derive design specifications and implementation. We highlight the significant parts of the B machine as follows.

**MACHINE SpyNetwork (maxSpies)**

The SpyNetwork machine has a set representing all the spies in the network. The machine invariant defines the types of the machine variables that represent the state of the machine. The machine variables include the set of spies as well as their security attributes such as their mailbox passwords, their public and private keys.

**VARIABLES**

- spies, spyld, spyName, spyMailBoxPassword,
- spyPublicKey, spyPrivateKey,
- key, signature — used as a placeholder for use in local variables

**INARIANT**

- spies : Spy & spyld : spies >>- NATURALI & spyName : spies >>- STRTOKEN & spyMailBoxPassword : spies >>- STRTOKEN & spyPublicKey : spies >>- STRTOKEN & spyPrivateKey : spies >>- (STRTOKEN * spyPublicKey) & spyld >>- spyName >> spyMailBoxPassword & spyPublicKey >> spyPrivateKey & spies >>- (NATURALI < STRTOKEN << STRTOKEN < STRTOKEN < STRTOKEN < STRTOKEN) & key : STRTOKEN & signature : STRTOKEN

The KAOS operations are mapped to the following B operations. We use the signRevelation and accessMailbox operations as examples of KAOS operations that are transformed to their B equivalents and refine them in B due to space limitation.

\[
\text{signature} \leftarrow \text{signRevelation(revelation, identity)} = \begin{cases} \text{PRE} & \text{revelation : STRTOKEN & identity : NATURALI} \text{ THEN} \\
& \text{IF (identity : ran(spyld)) THEN} \\
& \quad \text{key := spyPrivateKey(identity) ||} \\
& \quad \text{signature := revelation >>- STRTOKEN} \\
& \text{ELSE} \\
& \quad \text{signature := STRTOKEN} \\
\end{cases}
\]

\[
\text{accessAllowed} \leftarrow \text{accessMailbox(identity, password)} = \begin{cases} \text{PRE} & \text{identity : NATURALI & password : STRTOKEN} \text{ THEN} \\
& \text{IF (identity : ran(spyld)) THEN} \\
& \quad \text{IF (password == spyMailBoxPassword(identity)) THEN} \\
& \quad \quad \text{accessAllowed := TRUE} \\
& \quad \text{ELSE} \\
& \quad \quad \text{accessAllowed := FALSE} \\
& \text{ELSE} \\
& \quad \text{accessAllowed := FALSE} \\
\end{cases}
\]

4.4 Derivation of Design and Implementation

The abstract machine obtained from transforming the KAOS operations to B is then refined to formally derive design specifications and implementation of the SNS security requirements. Each B refinement step prior to the implementation refinement reflects some design decision(s) added by the refining B machine to the refined B machine until implementation is obtained. The refinement mechanism in B provides means for documenting design decisions and building forward traceability links from requirements to design. This is achieved through building a more detailed model from a more abstract one.

The first refinement step for the SNS focuses on data refinement through making the design decision of representing the pool of spies as an array of spies and we will show how the signRevelation and accessMailbox operations are refined accordingly as examples. From the traceability perspective, we can see that this data refinement step does not address the realization of a specific security requirement in the system. It rather concentrates on building a concrete data structure representing the internal system state in a form realizable by programming languages while implementation is generated.
The second refinement step makes a design decision of employing DSA (Digital Signature Algorithm) as the signature algorithm to achieve the revelation integrity requirements. This refinement step provides traceability links between integrity requirements and the design that realizes them. The second refinement step is classified as a procedural refinement since it only removes the non-determinism of the digital signature algorithm being used without modifying the state representation of the refined machine. The second refinement step is not described in detail due to space limitation.

The next step is to refine SpyNetworkR1 (the second refinement step) to the implementation machine.

**IMPLEMENTATION SpyNetworkR1**

**REFINES SpyNetworkR1**

**SEES StrTokenType, Bool_TYPE**

**IMPORTS**

SpyNetworkUtilities, privateKeyArray(maxSpies), publicKeyArray(maxSpies), spyMailboxArray(maxSpies)

**INARIANT**

spiesArray = spies & privateKeyArray = spyPrivateKeys & publicKeyArray = spyPublicKey & spyMailboxArray = spyMailboxPassword

**OPERATIONS**

signature := signRevelation(revelation, identity) =

VAR ii, privateKey, publicKey IN

ii := 0;

WHILE ii <= maxSpies

DO ii := ii + 1;

The final step is to generate C code from the implementation machine. The programming language choice is based on the programming languages available in the B tool being used. Almost all the commercial B tools generate code in C and very few of them generate ADA. The security properties should be maintained by the design decisions and the semantics of the B machines rather than by specific security constructs in the programming language to which the implementation machine is translated.

5. Security Specification Changes

With the introduction of security changes to software systems, more vulnerability might be added to the system as a result of the maintenance activities. Current security engineering approaches might not either consider maintenance activities or provide sufficient information to perform accurate impact analysis resulting in introducing security vulnerabilities. In this section, we demonstrate the
capability of FADES to structurally handle changes of security specifications by providing sufficient traceability information allowing for more complete and accurate impact analysis of change. According to [13], the use of a software model that stores design decisions and traceability links significantly improve the accuracy and completeness of impact analysis.

Maintenance activities are classified into four categories according to [5]: adaptive (changes in the software environment), perfective (new user requirements), corrective (fixing errors), and preventive (prevent future problems). We have chosen an example of a corrective change since corrective changes consume 21% of change requests [5]. The other three categories of maintenance activities will be part of our future work.

A defective scenario threatening revelation confidentiality is as follows: A spy leaves his team and gets reallocated to another team after a message has been sent. This scenario is not handled by the current encryption/decryption solution used to protect the confidentiality of revelations since the leaving spy would receive a revelation that he is no longer eligible to receive. To correct this defect, the KAOS framework provides a conflict construct that allows the expression of situations that contradict with system requirements. We introduce the following conflict to the RevelationConfidentiality goal.

Conflict KnowingAfterLeavingTeam
InformalDef A spy knows a revelation targeted to a team although he has left
FormalDef \forall sp1, sp2 :: Spy, te1, te2 :: Team, re :: Revelation
\exists Member(sp1, te1) \land Member(sp2, te1) \land \neg [Knows(sp2, re, Content) \\
\exists Member(sp2, te2) \land te1 = te2]

This conflict could be resolved using one of the patterns for conflict resolution [4] by introducing a new goal to anticipate the conflict:

InformalDef A team relay is notified one day before a spy in his team is reallocated
FormalDef \forall sp1, sp2 :: Spy, te1, te2 :: Team
Member(sp1, te1) \land Member(sp2, te2) \land te1 = te2 \land \neg [Knows(sp2, re, Content) \\
\neg Member(te2, Relay(sp1, te1))]
\exists Member(te1, Relay(sp1, te2))
\exists mn :: Member(te1, Relay(sp1, te2)) \land \neg Member(te1, Relay(sp2, te2))

This goal could be assigned to a reliable agent such as the big boss. Analyzing the impact of introducing the new goal shows that the RevelationConfidentiality requirement would be affected by this change. The traceability information provided by the hierarchical structure of the goal graph and the KAOS refinement mechanism direct the change impact analysis to revisit the AND refinement of the RevelationConfidentiality goal. The new goal needs to be added as a subgoal to the refinement of the RevelationConfidentiality goal. The goal graph for the RevelationConfidentiality goal would be modified as in Figure 4 (notice the dark gray subgoal that has been added to the refinement of the RevelationConfidentiality goal):

![Figure 4: Accommodating the New Goal](image)

Since the new goal is a leaf goal, it will be operationalized using the following operation:

**Operation** NotifyRelayWithReallocation

**Input** Spy{arg relay}, Spy{arg leavingSpy}

**DomPre** \((\exists relay, leavingSpy :: Spy) \land Notified(relay, leavingSpy)\)

**DomPost** \((\exists relay, leavingSpy :: Spy) \land Notified(relay, leavingSpy)\)

**ReqPreFor** \((\exists team1, team2 :: Team) \land Member(relay, team1) \land Member(leavingSpy, team1) \land 0 < 24h \land Member(leavingSpy, team2)\)

This operation needs to be transformed to B in order to propagate this corrective change to the derived design and implementation. According to the change impact analysis performed with respect to the transformation of the new operation to B, we discovered that the state representation (Variables) of the SpyNetwork machine needs to be complemented with the following variables: team, relaySpies, authorizedReceiversFrom, authorizedSendersTo. These variables represent the set of assigned relays of all teams as well as the set of spies authorized to send to or receive from relays. Constraints on these variables need to be added to the invariant of the machine as follows:

```
team : spies <-> STRTOKEN & relaySpies <= spyId & authorizedReceiversFrom : relaySpies -> spies & authorizedSendersTo : spies -> relaySpies
```

The definition of the operation notifyRelayWithReallocation is given below while its refinement is not shown here due to space limitation.
6. Conclusion and Future Work

This paper presented the FADES approach using the SNS case study. This research addressed a case study that represents a sample of the category of communication systems using the FADES approach. This category of systems share a common set of security requirements that have been elaborated in the paper. This assists in verifying the applicability of FADES to communication systems that exhibit high security demands. We have also illustrated how the underlying formality allows for documenting design decisions and trace these decisions to their corresponding security requirements.

While this paper demonstrated the conceptual framework of FADES, a number of future research steps are ongoing. We have embarked on case studies from other categories of software systems for further refinement and to show its applications. Further, FADES is being quantitatively compared to other relevant approaches that derive design from requirements. The comparison will involve the application of both FADES and the other comparable approaches to the same case studies to measure the strength and limitations of FADES as opposed to other relevant approaches.

Future work might involve the demonstration of the capability of FADES to structurally handle other categories of maintenance activities (perfective, adaptive and preventive).

Acknowledgement

This work was supported in part by Science Foundation Ireland grant 03/CE2/I103_1 to Lero-the Irish Software Engineering Research Center (www.lero.ie).

Thanks to Dr. Van Lamsweerde who granted us a free license of the Objectiver software that has facilitated our work in the elicitation and modeling of security requirements using the KAOS framework.

References