Technical Report

Nr. TUD-CS-2014-0865
July 14th, 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

EC SPRIDE
EUROPEAN CENTER FOR
SECURITY AND PRIVACY BY DESIGN

Using Assurance Cases to Develop Iteratively Security Features Using Scrum

Authors
Lotfi ben Othmane, Fraunhofer SIT, Germany
Pelin Angin, Purdue University, USA
Bharat Bhargava, Purdue University, USA

Using Assurance Cases to Develop Iteratively Effective Security Features

***Abstract*—A security feature is a customer-valued capability of software for mitigating a set of security threats. Incremental development of security features, using the Scrum method, often leads to developing ineffective features in addressing the threats they target due to factors such as incomplete security tests. This paper proposes the use of security assurance cases to maintain a global view of the security claims as the feature is being developed iteratively and a process that enables the incremental development of security features while ensuring the traceability of tests to security requirements such that it ensures the effectiveness of implemented features.**

I. Introduction

A *security feature* is a customer-valued functionality and/or property of software for mitigating a set of threats. Examples of security features include controlled access to a database, authentication to use resources, and secure communication using Secure Sockets Layer (SSL).

There are three main risks in managing innovative software projects which apply to developing security features: limited knowledge about the technology needed to implement the feature, change of customer needs, and impact of other software quality aspects, such as performance. Innovative software development projects often include the use of new methods and technologies not well known by the developers. The developers gain the required knowledge through research and developing prototypes. For instance, they often use code libraries that are not well documented. This limits the practicality of the design since they could encounter incompatibilities of libraries, absence of assumed functionalities (e.g., preferred encryption algorithms), and interoperability of data structures–e.g., key-store file structure.

Developers often estimate the load of the software and deduce performance parameters, such as response delay and processing time, during development. In some cases performance measurements impact the design choices of security mechanisms. For instance, assume that the developers implement a protocol for communicating a concentrator that serves 50 devices with a remote application, which encrypts each message using the RSA algorithm. The processing requirements of RSA may prohibit the use of the same concentrator to serve 1000 devices. This scaling up requires the adoption of an encryption algorithm that requires shorter processing time.

As software evolves, its security requirements may change, which may require extending existing security features. For example, consider that a software that uses secure communication between a device and a remote Web application feature stores the private keys used by the feature in the flash memory of the device. The customer of the software wants to enforce the liability of the users through the use of smart cards included in the device for private key storage. The implementation of the requirement requires extending the security feature.

Controlling these risks favor the use of an iterative and incremental approach for developing security features. Companies commonly use Scrum [18], [19] to extend software including their security features. Scrum is an iterative and incremental software development method that enables producing potentially shippable software in successive iterations, each lasting a few weeks. However, the incremental and iterative nature of Scrum increases the complexity of ensuring that the implemented security features are effective in mitigating the threats they target, because code changes in the features could affect the efficiency of the mechanisms they implement.

Errors in implementing security features, which are frequent, make them ineffective. For example, Georgiev et al. tested a set of applications including known payment systems and mobile applications that use SSL to mitigate Man-in-the-Middle attack (MITM) and found that the implemented security features are ineffective [12].[1] Fahl et al. interviewed the developers of mobile applications that implement ineffective secure communication features [9]. Among the main causes that they found is incomplete security tests, that is, the errors could have been avoided if complete tests were used.

This paper aims to address the following questions: How can we ensure completeness of security tests and ensure effectiveness of a security feature? And how can we incrementally develop a security feature efficiently?

We address the first question by using security assurance cases, which maintain a global view of the security of the security features. We address the second question through a process for developing security features that ensures the effectiveness of the features in mitigating the threats they target. The process supports experimental learning to address potential challenges at each increment, enables developing increments in each iteration, and prevents redeveloping security mechanisms.

The paper is organized as follows: Section II discusses related work, section III describes the use of security assurance cases for iterative development, section IV describes the method we propose to develop security features, section V illustrates the use of the proposed method using the case study secure communication between a mobile device and a remote service, section VI discusses the proposed approach, and section VII concludes the paper.

II. Related work

Methods for developing secure software using the model driven approach have been very popular in the literature. For

[1] The paper does not specify explicitly that the features are developed iteratively, however, the authors learned that that's the case for many of them.

example, Basin et al. [3] proposed a method that involves the specification of system models along with their security requirements and the automatic generation of system architectures from the models. A second example is extending modeling languages, such as Unified Modeling Language (UML) to include system misuse cases during system requirements specification [10].

Developing secure software that includes security features using iterative development methods, such as Scrum is increasingly getting attention. For instance, Peeters [16] proposes the introduction of abuser stories in the requirements domain of agile development, which extend user stories to achieve traceability of security requirements. Abuser stories are ranked based on their value (perceived damage and likelihood of successful attack) and the amount of implementation effort they require, much like the ranking of user stories.

Bostrom et al. [6] propose a way of extending Extreme Programming (XP) to help developers and customers to engineer security requirements while maintaining the iterative feature of XP. Their approach extends the XP planning game process of identifying business requirements to integrate two kinds of new user stories into the development process: threat scenarios (abuser stories) and security functionalities that are used to address the identified threats. The selection of threat scenarios to address is based on defining critical system assets and risk assessment of abuser stories by calculating probability of occurrence.

The Open Web Application Security Project (OWASP) [14] proposes developing evil user stories (hacker abilities to compromise the system), and expressing them in a conversational style. An example for an evil story is: As a hacker, I can send bad data in URLs, so I access data for which I'm not authorized. The stories address authentication, session management, access control, input validation, output encoding/escaping, cryptography, error handling and logging, data protection, communication security, and HTTP security features.

Recently, Asthana et al. [2] proposed 36 generic security-focused user stories and 29 security tasks. The security-focused user stories are derived from the issues commonly seen by the authors, from the CWE/SANS Top 25 [7] most dangerous development errors, and the OWASP top 10 list [15]. The security tasks include 12 operational security tasks that needs to be performed by the development team in each sprint and 12 security tasks that could be performed as needed with the guidance of security experts, especially at the beginning of the project development.

These methods support considering security needs in the software development life cycle but do not consider the security assurance of produced software. This issue has been initially investigated by Beznosov and Kruchten [5] who identified the conflicts between agile development practices and security assurance techniques, and proposed strategies to address the conflict. Although the authors provide insights about the invention of new agile-friendly methods, they do not mention any details regarding the implementation of such methods.

Recently, Ben Othmane et al. [4] proposed a method for security reassurance of software increments in agile development that integrates security engineering activities into the agile development process. The method supports developing acceptably secure software at each development iteration through the use of an incremental approach of security assurance.

Our approach in this paper differs from previous approaches in that it uses security assurance cases to build security features incrementally, and maintains a global view of how each new increment of a software system affects the validity of security assurance cases from the previous iteration. By maintaining this view, it provides efficiency in security assurance, as it obviates the need to do a complete security assessment at each iteration.

III. Iterative security assurance cases

This section provides an overview of security assurance cases, describes our proposed method for using security assurance cases for iterative development and discusses the use of security assurance cases for traceability.

A. Overview of security assurance case

A *security assurance case* [22] is a semi-formal approach for objectively supporting the claim that a software product mitigates its security risks [4]. It is a collection of security-related claims, arguments, and evidences. A *claim* (i.e., a security goal) is a high-level security requirement. An *evidence* is the result of a claim's verification through, for example, security tests, source code security review, and proofs. An *argument* is a justification that a set of evidences support the related claim.

A security assurance case has a tree structure, where the root is the main claim, intermediate nodes are either sub-claims or arguments, and the leaves are the evidences. A common way to represent assurance cases is to use the Goals-Structuring Notations (GSN) [13]. *Goals-Structuring Notations* is a graphical argumentation notation that represents the elements of the assurance case and the relationships between these elements.

The main steps of creating a security assurance case, in sequence, are [4]:

1) Decompose the claim "the software is secure" into sub-claims such that satisfying the sub-claims induce satisfying the claim. The sub-claims (which in turn become claims) could be iteratively subdivided, until getting verifiable sub-claims.
2) Specify the context of the claims, such as definitions, reference to documents, explanations, and assumptions.
3) Identify the strategies for decomposing the claims into sub-claims. The strategy could be explicitly described in the assurance case or be implicit if no strategy is specified.
4) Collect the evidences that support the claims, which are the results of using the security evaluation techniques, such as security testing and security review of source code to evaluate the security countermeasures [20] used to eliminate or reduce the risk of the threats to the software and achieve the related security goals.
5) Describe the arguments that show that an evidence supports a claim. For example, the results of a security analysis tool may report that the software has a buffer overflow, but the argument could state that the "errors" are false positives–so the claim is satisfied.

B. Incremental security assurance cases

Changes to the software increments (the software produced by an iteration and includes software produced by the previous iterations) affect the security assurance case in four ways, which are [4]:

- Component updates could invalidate evidences and claims associated with the component and could potentially affect claims associated with components related to the modified component.
- Adding a new component requires reevaluating the claims related to the new component and the claims that could be affected by updating other components connected to the new component.
- A change of context of using the software requires reevaluation of the related claim and other associated claims.
- The set of claims that need evaluation due to adding a new claim includes the new claim, sub-claims, and parent claim, i.e. related claims.

C. Use of security assurance case for traceability

Traceability is the ability to describe and to follow the life-cycle of the development artifacts [23], i.e., requirements, design, code, and test results. It is commonly used to ensure the maintainability of software by establishing relations between the development artifacts. It is used to estimate the impacts of changes, and to detect inconsistency and incompleteness of the development artifacts [23].

A security assurance case relates security claims and security assessment results (e.g., security tests and security code analysis results). It supports traceability of security development artifacts. We use security assurance case as a traceability technique, but to support changes to security artifacts–not functional artifacts.

IV. Proposed iterative process for developing security features

This section gives an overview of iterative development using Scrum, motivates the proposed method, and provides a description of the proposed method.

A. Iterative development using Scrum

The *Scrum*[2] method was discovered by Takeuchi and Nonaka as a practice by several companies to develop innovative products [21]. The method is commonly used to iteratively develop software in increments.

The Scrum development process is composed of a controlled set of loose activities [18]. The three phases of Scrum are pregame, game, and postgame. The *pregame* phase, or project inception, serves for planning and designing the system at a high level. The *game* phase, or construction phase, serves for developing the user stories. The *postgame* phase, or closure phase, serves for preparing the increment for release to customers.

[2] a formation for forwards binding in three rows.

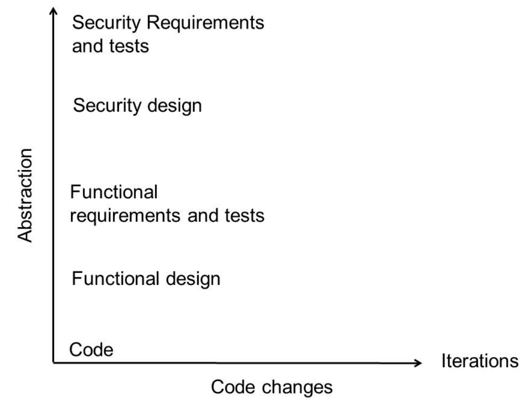

Fig. 1. Evolution of security features.

B. Motivation for the proposed method

A security feature implements a set of system functionalities that mitigates a set of security threats. Figure 1 relates the development artifacts of a security feature to code changes. The development artifacts of a security feature ordered by level of abstraction are: security requirements and related test scenarios, security design, functional requirements and related test scenarios, functional design, and code that implements the feature. Changes to a security feature requires revising all the development artifacts to maintain the effectiveness of the feature.

There are two commonly used approaches for developing security features: the top-down approach and the bottom-up approach. The top-down approach requires designing a solution for the feature as a set of collaborating components. The approach (1) disregards the value of feedback from the business owner, and (2) assumes that the developers have full knowledge about how to implement the feature. These issues cause the developers to potentially fail the project due to (1) unknown technical barriers (e.g., incompatibility of libraries) that they could have avoided if they had the chance to develop experiments and adapt the design according to the results and (2) absence of mechanisms to adapt the feature according to the feedback about the business context of using the feature.

The bottom-up approach is based on developing a simple experiment and iteratively increasing its complexity to finally have the full security feature. This approach potentially leads to (1) redeveloping existing features and (2) implementing insecure solutions that address only immediate needs. These issues could waste developers' time on developing unnecessary components–e.g., encryption and signature components–that could be avoided by using existing libraries, and result in developing partial solutions that may have severe limitations. For example, implementation of a communication protocol between a mobile application and a remote service using a symmetric key cannot be used by a set of mobile phones; having a key shared by a set of parties limits confidentiality of the exchanged messages–all parties can decrypt the messages–through extending the feature from being used by one mobile phone to being used by a set of phones.

In this paper we propose an approach that favors the

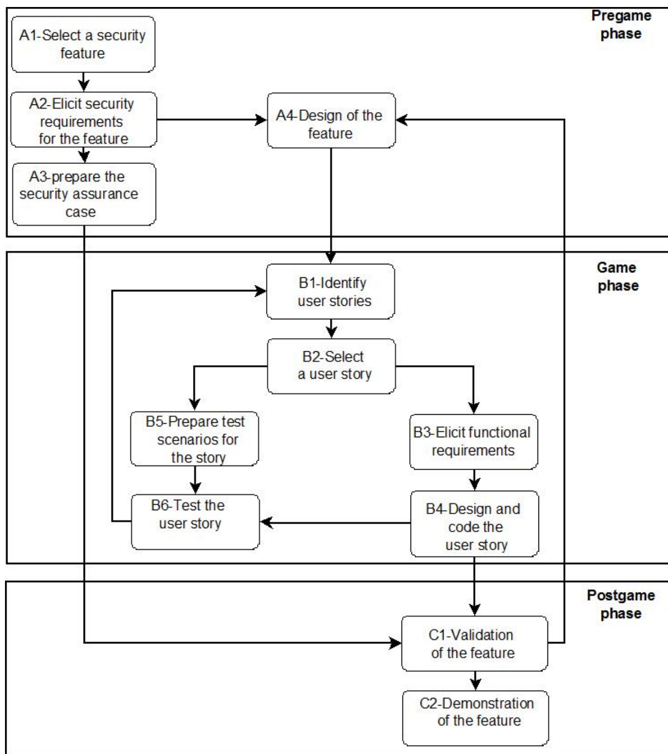

Fig. 2. The proposed iterative process for developing security features.

design of security features and incremental increase of the complexity of implementation. The main steps are: design a high-level architecture of a security feature, set a road map for implementing the feature through incremental increase of the complexity of the solution, and iteratively develop the increment while applying lessons learned to address challenges in subsequent iterations.

The approach enables developing increments in each iteration and prevents redeveloping security mechanisms. The incremental increase of the software complexity (that we propose) is different from the components-based method used in the top-down approach. The latter favors developing the components individually and integrating them. The former favors updating all the components of the software for each iteration–as needed–such that they implement together the given user stories.

C. Proposed development process.

We propose to extend the three phases of the Scrum process–i.e., pregame, game, and postgame–such that we ensure iterative development of security features that effectively address the threats that they are designed to mitigate. Figure 2 shows the phases and the activities for each phase. The description of this process, which should be used for each security feature, follows.

Pregame Phase. The goal of this phase is to identify sufficient information and make the main design decisions for developing the selected security feature. First, the product owner and developers *select a security feature* to develop (i.e., activity A1) among the security features based on factors such as the priority of the feature, availability of resources, cost, and release date.

Next, the developers *elicit the security requirements of the feature*; that is, what the feature should do and not do. Examples of security requirements for a secure communication feature include preventing unauthorized disclosure of data, preventing unauthorized modification of data, using a strong secret key, and protecting the system from Denial of Service (DoS) attack exploiting TCP handshaking vulnerabilities.

Then, the developers *design the feature*. They evaluate existing approaches that address similar problems from the literature;[3] design a set of functionalities that implement the security requirements; identify the technical constraints related to implementing the design including the choice of programming language, available libraries and frameworks, and processing and memory constraints; and select algorithms and software packages that will be used to develop the feature.

After that, they *prepare the security assurance case* of the feature. They enumerate the claims associated with the feature and design attack scenarios (a sequence of steps that exploit vulnerabilities of the system and cause threats to the software) that "verify" the claims. An example test scenario for the secure communication between a device and a remote server is letting a device communicate with a remote server using an expired digital certificate.

Game Phase. The goal of this phase is to implement the feature. The developers *identify a set of user stories* that implement the design. A *user story* describes a functionality valuable to the user of the software [8]. The user stories must enable the *incremental* development of the feature by increasing the complexity of the software.

Some stories have dependency relations with other stories and others do not. Incremental development of the user stories that implement a security feature requires *selecting user stories* (Activity B2) whose dependent user stories have been implemented in previous sprints.

Then, the developers *elicit the functional requirements for the selected user story* (Activity B3). They identify *what* the software must do such that it mitigates the threats that the feature addresses. For example, device certificates must be signed using the system's private key for signing certificates, generated keystore and truststore must authenticate the specific device, and generated keystore and truststore must be compatible with the protocol.

Next, the developers *design and code the user story* (Activity B4). They develop algorithms and mechanisms that implement the functional requirements and identify a set of tasks for changing the current software by e.g., adding new methods and classes that implement the user story. Note also that implementing a single user story may affect several components of the system at the same time.

After selecting the user story to implement, the team members proceed with *preparing tests for the user story* (Activity B5). They develop test scenarios that test the functional

[3] We assume there are solutions in the literature that address the security feature.

requirements and specify the expected results.[4] A detailed example of a test scenario is: update the file `Mydevices.txt` and execute the script `genCrypto.py`. The script creates a folder and generates a keystore file and a certificate for each device listed in the file `Mydevices.txt`. Note that this test scenario does not test the validity of the generated certificates, which could be the subject of another test scenario.

After completing the development of the user story and the preparation of the test scenarios, the developers *perform the test scenarios* (Activity B6). They execute the test scenarios for a set of selected cases and compare the obtained results to the expected results. If the test results are positive they conclude the user stories and proceed to selecting another user story (Activity B2). If the tests are not conclusive the developers identify how to address the differences. They have mainly two choices: (1) review the design and code to fix the problem or (2) identify new user stories that address the problem. The team demonstrates the user story at the end of the sprint by showing the successful test scenarios.

Postgame Phase. The goal of this phase is to prepare the feature for use in a release. After developing (and testing) the user stories that implement the feature, the team proceeds with *testing the feature* (Activity C1). They execute the security test scenarios defined in Activity A3 and compare the obtained results to the expected results. If the obtained results mismatch the expected results, the developers review the design of the feature (Activity A4). Otherwise, they proceed with *demonstrating the feature* (Activity C2) to the product owner. They explain the value of the feature and why it mitigates the threats, and demonstrate a set of security scenarios on a variant of the system that does not have the feature and on a variant that has the feature. Then, they update the security assurance case with the test results.

V. Case study: Secure communication between a mobile device and a remote service

In this section we illustrate the use of the proposed process and iterative security assurance method to iteratively develop the security feature for secure communication between a device and its remote control service. Figure 3 shows an example of systems where a device installed in an unmanned aerial vehicle (UAV) communicates with a remote control station.

We simulate the implementation of the feature in two releases; each is developed using a set of user stories. The descriptions of the releases follow.

First release: The goal of this release of the feature is to "secure" the communication between the devices and the control service. Table I lists a set of security requirements, decision choices, and test scenarios for the feature[5] identified during the *game phase*. The security requirements become the claims of the assurance case of the feature as illustrated in Figure 4.

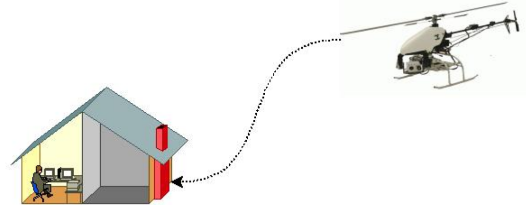

Fig. 3. A remotely controlled unmanned aerial vehicle. (The UAV is a Cadence Technologies SR-20.)

TABLE I. Example of security requirements, design decisions, and test scenarios for Release 1 of the security feature "secure communication between the devices and control service."

	Main requirements
SR1	Protect the confidentiality of data exchanged between the device and remote control service
SR2	Protect the integrity of data exchanged between the device and remote control service
SR3	Each device must authenticate the remote control service before sending data.
SR4	Use of cryptographic keys and algorithms that are secure for the next 10 years.
SR5	The service authenticates the devices it communicates with.
	Main design decisions
SD1	Use SSL protocol [11] for secure communication between two parties.
SD2	Use SSL libraries JESS for the remote service and JESSIE for the device. (The decision is made considering the system architecture of the system.)
	Test Scenarios
	Intercept communication between a testing device and remote service using Wireshark [1] and perform the following tests:
STS01	Test if the data in transit between the device and service are plain text.
STS02	Test if changing the data in transit between the device and service could be detected.
STS03	Test if the device sends data to a service without successfully authenticating it.
STS03a	Authenticate a device using a signed certificate stored in its keystore.
STS03b	Authenticate a device using an unsigned certificate stored in its keystore.
STS03c	Authenticate a device using an expired certificate stored in its keystore.
STS04	Test if the service sends data to a device without the device successfully authenticating it.

The team may consider other requirements, besides the security requirements, in the design of the solution for the security features. For instance, examples of requirements that are often considered for the feature of secure communication between two parties are optimizing the use of bandwidth and optimizing the use of device processing time for security operations.

Next, the developers start the *game phase* and enumerate foreseen user stories required to implement the security feature, which we list in Table II. We note that the user story

TABLE II. List of user stories for the first release of the security feature.

Code	User stories
R1U1	As a user, I can securely send data from a desktop to a server application.
R1U2	As a user, I can securely send data collected by the device to a server application.
R1U3	As an administrator, I can identify the device that sends a particular set of data.
R1U4	As a user, my device can store the data when it cannot connect to the server and send it when communication is restored.
R1U5	As administrator, I can create a set of private/public keys pairs that are associated with the identities of the devices in batches.
R1U6	As administrator, I can easily distribute each private/public key pair to the associated device.
R1U7	As administrator, I can create for each device a set of private and public keys, each constructed using the identity of the device.

[4]Recall that a security test scenario is a set of steps to attack the system and cause a threat. Differently, a functional test checks if the system performs a function as expected. For example, a functional test is: The system checks digital certificate of communicating partners, while a security test is: The device authenticates with the remote service using an expired certificate.

[5]The list is only to illustrate the method and is not exhaustive, nor detailed.

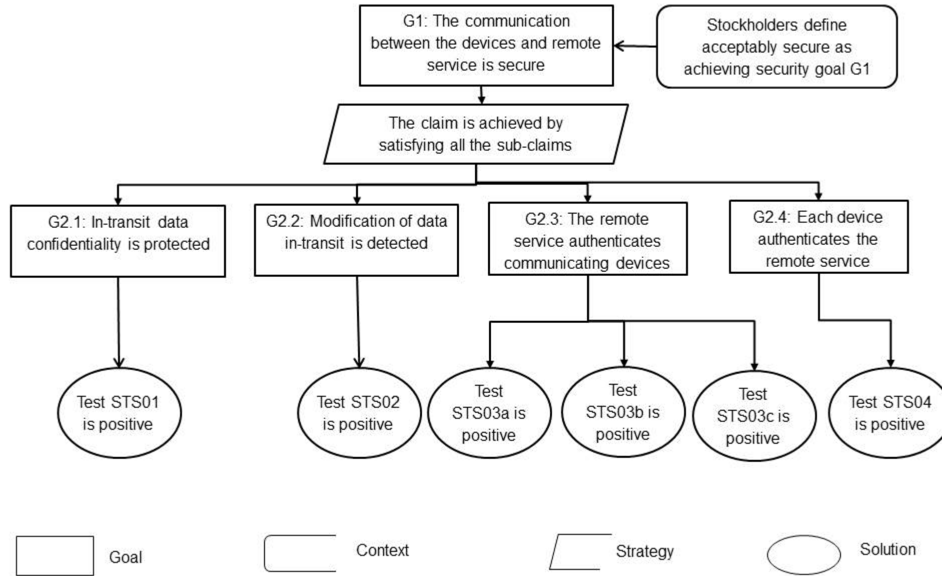

Fig. 4. Security assurance case for Release 1 of the security feature. (The code of the tests are in table I.)

TABLE III. EXAMPLE OF FUNCTIONAL REQUIREMENTS, DESIGN DECISIONS, AND TEST SCENARIOS FOR THE USER STORY R1U5

Code	Main requirements
FR1	Generate a keystore and truststore for each device given a list of device identities.
FR2	The certificate of each device must be signed using the company private key.
FR3	Each certificate must be associated with a single device.
FR4	The generated trustore and keystore files must be compatible with the device software.
FR5	Use of cryptographic keys and algorithms that are secure for the next 10 years.
FR6	The service authenticates the devices it communicates with.
	Main design decisions
FD1	Use a Python script to automate the generation of private keys and a certificate and prepare them for use by the devices and service.
FD2	Use a text file that lists the identities of the devices to generate the private keys and certificates.
	Test Scenarios
FTS1	Generate a keystore and trustore for a set of devices.
FTS2	Authenticate a device using a signed certificate stored in its keystore.
FTS3	Reject authentication of a device using an unsigned certificate stored in its keystore.

$R1U7$ does not depend on the user story $R1U1$. Therefore, it does not need to be developed after $R1U1$. However, user story $R2U1$ depends on $R1U1$ because $R1U2$ increases the complexity of the software.

After that, the team selects successive user stories to develop considering the dependencies between them. For each selected user story they elicit the security requirements, design a solution that implements the user story, identify a set of test scenarios to test it, and develop the feature and evaluate it using the test scenarios. For example, Table I lists the user story $R1U5$: as administrator, I can create a set of private/public keys pairs that are associated with the identities of the devices in batches.

At the *postgame phase*, the team performs the security tests and updates the security assurance case with the evidence collected from the tests as in Figure 4. Then, they demonstrate the feature in the retrospective meeting with the product owner. The retrospective meeting is an *end of iteration* meeting to discuss the successes and the challenges encountered in the iteration and to set changes that enables to address the challenges, such as requesting trainings, changing the development process, and acquiring new tools that help in addressing the challenges.

Second release: In the first release the private keys are stored in the flash drive of the device. Let's assume that data sent by the devices is required to be used in a legal context, which requires the system to ensure non-repudiation of the sender. The product owner requests to upgrade the feature to satisfy the new requirement.

At the *pregame phase* of this release, the team members review the security requirements, decisions and test scenarios. Assume they decide to use a smart card embedded in the device to store the cryptographic keys and use it to perform computation using these keys. Therefore, the team reviews the security assurance case of the feature. Since the design decision requires using new components for the cryptographic operations, the evidences of the security assurance case become invalid and are removed from the case of the feature.

Then, the team starts the *game phase* and identifies the user stories required to implement the security feature, which are listed in Table IV. Next, they proceed with iteratively implementing the user stories as in the first release–considering the dependencies between them.

At the *postgame phase*, the team performs the security test scenarios and updates the security assurance case with the evidence collected from the tests. Figure 5 shows an updated

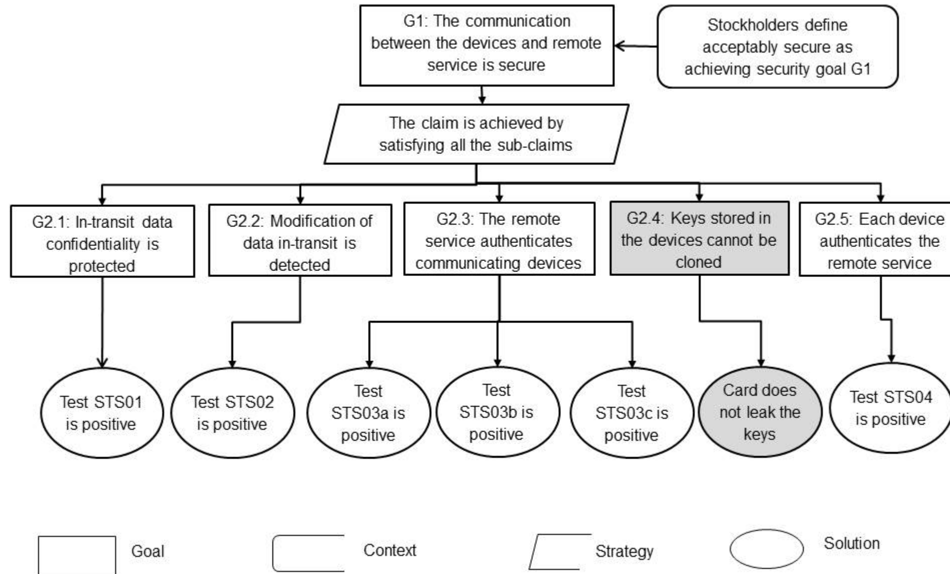

Fig. 5. Security assurance case for Release 2 of the security feature. (The code of the tests are in table III.)

TABLE IV. LIST OF USER STORIES FOR THE SECURITY FEATURE SECURE COMMUNICATION BETWEEN DEVICES AND REMOTE SERVICE FOR RELEASE 2.

Code	User stories
R2U1	As a user, I want the smart card of the device to perform the cryptographic functions required to communicate the device and remote service.
R3U2	As a user, I want the device to use the smart card to perform the cryptographic functions required to communicate the device and remote service.
R4U3	As a user, I want the device to use the smart card for cryptographic operation in its communication with the remote service.

security assurance case of the feature. The gray shapes indicate components added to the case in the second release.

VI. DISCUSSION

This section discusses how the proposed process enables iterative development of security features and traceability of artifacts and how iterative assurance cases help to have better coverage of test scenarios. It also discusses the limitations of the method.

Iterative development. We applied the approaches discussed in Section IV-B in developing a commercial project. We learned that the causes of the failures are limited knowledge about the technology needed to implement the feature, the change of the customer needs, and impacts of other software quality aspects, such as performance. Then, we applied the proposed iterative development process to develop the first iteration of the feature.

The proposed process enables iterative development at two levels: product level and release level. The product level enables to adapt the coherent design of the given security feature that addresses a set of security requirements for a given release such that it supports the security requirements of a new release. The release level enables the controlled implementation of the feature such that it is possible to adapt the functional design of the feature and the development plan to address the challenges as they are discovered.

The proposed iterative process uses two decomposition levels: property level and functional level. The property level is concerned with the security requirement, design and test scenario. The functional level concerns the functional requirements, design, and test.

The Scrum process is commonly known for functional features. The proposed iterative process extends the Scrum process to support security features which are properties of the system–not functions.

Traceability of artifacts. Traceability in the context of iterative development has two dimensions: vertical traceability that relates development artifacts of the same increment and horizontal traceability that relates iterations [23]. The proposed process enables traceability of the development artifacts at both the product level, i.e., horizontal traceability, and release level, i.e., vertical traceability. For each release of the product, the security test scenarios are related to the security design and to security requirements. Also, for each iteration of a given release, the functional requirements could be traced to specific security design and the functional tests could be traced to functional design and functional requirements.

The use of security assurance cases helps to manage the impacts of changes on the security of the software. It allows to identify–manually–invalidated evidences and related claims and to detect inconsistency and incompleteness of the evidences, arguments, and claims.

Test scenarios coverage. When we applied the development

process to develop and test the second release of the feature in the commercial project (see case study) we faced the challenge of identifying the evidence and claims that need to be reevaluated. We found that the security assurance case technique can address the problem. We did not practice with the technique because the project ended at that time.

The tree structure of security assurance cases supports identifying "potentially" the exhaustive list of the test scenarios that assert the security claims of the case. The tree structure enables dividing each node that represents a concept, such as attack or claim, into sub-claims and helps to visually identify the completeness of the sub-claims. This ability is used by the attack tree technique [17] to help identify all potential attacks. Security assurance cases also help solve conflicts of requirements identified in different iterations and releases.

Limitations. The proposed method does not help to discover vulnerabilities in libraries used by the features. Therefore, the method may allow to claim that a feature is secure (since the security test scenarios cover the security requirements), which could be challenged because it uses a set of libraries that have vulnerabilities that were not identified by the security tests. This is considered a change of the scope of the feature.

We currently do not have tools that support the application of the method. We hope to build such tools in the future.

VII. Conclusions

Using iterative methods, such as Scrum to develop security features often leads to developing ineffective features because of the incompleteness of the test scenarios and the conflict of requirements between the iterations.

This paper proposes a process for iteratively developing security features and a technique for ensuring that the feature is effectively secure. The proposed process was developed in response to several failures and it addresses the challenges of limited knowledge about the technology needed to implement the feature, change of the customer needs, and impact of other software quality aspects, such as performance.

The method supports developing effective security features. The use of the iterative security assurance case technique helps identify all security test scenarios and it addresses the conflicts between the security requirements identified in different releases and functional requirements identified in different iterations.

References

[1] Wireshark. go deep. Accessed on Mar. 2014. [Online]. Available: http://www.wireshark.org/

[2] V. Asthana, I. Tarandach, N. ODonoghue, B. Sullivan, and M. Saario, "Practical security stories and security tasks for agile development environments," Online, July 2012. [Online]. Available: http://www.safecode.org/publications/SAFECode_Agile_Dev_Security0712.pdf

[3] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security: From uml models to access control infrastructures," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 39–91, Jan. 2006. [Online]. Available: http://doi.acm.org/10.1145/1125808.1125810

[4] L. ben Othmane, P. Angin, H. Weffers, and B. Bhargava, "Extending the agile development approach to develop acceptably secure software," *Dependable and Secure Computing, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[5] K. Beznosov and P. Kruchten, "Towards agile security assurance," in *Proc. of the 2004 Workshop on New Security Paradigms*, ser. NSPW '04, Nova Scotia, Canada, Sep. 2004, pp. 47–54.

[6] G. Boström, J. Wäyrynen, M. Bodén, K. Beznosov, and P. Kruchten, "Extending xp practices to support security requirements engineering," in *Proceedings of the 2006 international workshop on Software engineering for secure systems*. ACM, 2006, pp. 11–18.

[7] S. Christey, B. Martin, M. Brown, A. Paller, and D. Kirby, "Cwe - 2011 cwe/sans top 25 most dangerous software errors," Online, Sep. 2011. [Online]. Available: http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf

[8] M. Cohn, *User Stories Applied: For Agile Software Development*. Boston, MA: Pearson Education, Inc., 2004.

[9] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516655

[10] I. Flechais, M. A. Sasse, and S. M. V. Hailes, "Bringing security home: A process for developing secure and usable systems," in *Proceedings of the 2003 Workshop on New Security Paradigms*, ser. NSPW '03. New York, NY, USA: ACM, 2003, pp. 49–57. [Online]. Available: http://doi.acm.org/10.1145/986655.986664

[11] A. Freier, P. Karlton, and P. Kocher, *The Secure Sockets Layer (SSL) Protocol Version 3.0*, Internet Engineering Task Force (IETF) Std., Aug. 2011. [Online]. Available: http://tools.ietf.org/html/rfc6101

[12] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating ssl certificates in non-browser software," in *Proc. of the ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012, pp. 38–49.

[13] T. Kelly and R. Weaver, "The goal structuring notation–a safety argument notation," in *Proc. Dependable Systems and Networks–Workshop on Assurance Cases*, Florence, Italy, July 2004.

[14] OWASP, "Agile software development: Don't forget evil user stories," August 2011. [Online]. Available: https://www.owasp.org/index.php/Agile_Software_Development:_Don%27t_Forget_EVIL_User_Stories

[15] ——, "Owasp top 10-2013 – the ten most critical web application security risks," Online, December 2013. [Online]. Available: http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf

[16] J. Peeters, "Agile security requirements engineering," in *Symposium on Requirements Engineering for Information Security*, 2005.

[17] B. Schneier, *Secrets and Lies*. John Wiley and Sons, New York, 2000, ch. Attack Trees, pp. 318–333.

[18] K. Schwaber, "Scrum development process," in *Proc. of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, R. Wirfs-Brock, Ed., Austin, TX, Oct. 1995, pp. 117–134.

[19] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[20] R. Shirey. (2007, aug) Internet security glossary, version 2. RFC 4949 (Informational). [Online]. Available: http://www.ietf.org/rfc/rfc4949.txt

[21] H. Takeuchi and I. Nonaka, "The new product development game," *Harvard Business Review*, vol. 64, no. 1, Jan 1986.

[22] J. Vivas, I. Agudo, and J. Lpez, "A methodology for security assurance-driven system development," *Requirements Engineering*, vol. 16, no. 1, pp. 55–73, 2011.

[23] S. Winkler and J. Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Softw. Syst. Model.*, vol. 9, no. 4, pp. 529–565, Sep. 2010.