Abstracting security-critical applications for model checking in a model-driven approach

Marian Borek, Kurt Stenzel, Kuzman Katkalov, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Borek, Marian, Kurt Stenzel, Kuzman Katkalov, and Wolfgang Reif. 2015. "Abstracting security-critical applications for model checking in a model-driven approach." In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), 23-25 September 2015, Beijing, China*, 11–14. Piscataway, NJ: IEEE. https://doi.org/10.1109/icsess.2015.7338996.

Abstracting Security-Critical Applications for Model Checking in a Model-Driven Approach

Marian Borek, Kurt Stenzel, Kuzman Katkalov and Wolfgang Reif
Department of Software Engineering
University of Augsburg, Germany
{borek,stenzel,katkalov,reif}@informatik.uni-augsburg.de

***Abstract*—Model checking at the design level makes it possible to find protocol flaws in security-critical applications automatically. But depending on the size of the application and especially on the abstraction of the application model, model checking may need a lot of resources, primarily time. To reduce the complexity, the application models are usually highly abstracted. But in a model-driven approach with automatic generation of runnable applications the application models need to be detailed and are often too complex to check in reasonable time. In this paper we describe an approach to handle this problem by using additional UML models to restrict the protocol runs, the attacker abilities and the numbers of participants. This makes model checking of large applications in our model-driven approach called SecureMDD possible without manual abstraction of the generated specifications. For model checking we use AVANTSSAR and show how the restrictions modeled within UML are translated. We demonstrate our approach with a smart card based electronic ticketing example.**

***Index Terms*—UML; model checking; security-critical systems; model-driven development; transformations; SecureMDD**

I. Introduction

Security-critical applications use security protocols that can be application-independent like TLS but also application-specific like in TextSecure and SnapChat. For both kinds of security protocols the development is difficult and error-prone [16], [11]. But application-specific protocols have to be developed and analyzed for each new application. SecureMDD is a model-driven approach to develop security-critical applications and their security protocols. It supports interactive verification and model-checking of application-specific security properties [8] that are more fine-grained than standard security properties like integrity or confidentiality and are very close to real security requirements. But especially for model checking of large security-critical applications there is a need for abstractions to make model checking feasible [7]. The main purpose of model checking of security-critical applications is to find security flaws. With the knowledge that model checking needs abstractions and each abstraction can cause that an attack can not be found our goal is to provide a clear and easy mechanism to define and alter abstractions for model checking.

In our previous works we described two model driven testing approaches for security-critical applications. We are able to model concrete attacks and generate test code for the real application [14]. In addition, it is also possible to transform a modeled application into a formal specification for model checking [7]. Both approaches have their strengths and weaknesses. The first approach is very useful for quick regression testing but the attacks have to be specified manually. The second approach finds attacks automatically by model checking, but needs a lot of resources like CPU power and time. In this paper we combine the advantages of both approaches by restricting the complexity of the model but without specifying a concrete attack.

This paper is structured as follows. Section 2 gives a short overview of our model driven approach called SecureMDD and section 3 depicts an electronic ticketing example that is used as a case study in this paper. Section 4 describes the modeling guidelines to restrict the complexity for model checking with UML diagrams and section 5 depicts the specifications generated from the modeled restrictions. Section 6 compares the results between automatic and manual abstractions and section 7 discusses related work. Section 8 concludes this paper.

II. The SecureMDD Approach

SecureMDD [15], [6] is a model driven approach to develop secure applications. From a UML application model using a predefined UML profile and a platform-independent and domain-specific language (MEL) runnable code for different platforms (e.g., Java Card for smart cards, Java for user devices, and Java based Web services for services) as well as formal specifications are generated. One formal specification is used for interactive verification with KIV [10], [4] and the other to find vulnerabilities with the model checker platform AVANTSSAR[1]. The static view of an application is modeled with UML class diagrams and deployment diagrams. The dynamic behavior of system components is modeled in UML activity diagrams with MEL. The approach is fully tool-supported and all model transformations are implemented. For further information visit our website[1].

III. Case Study: eTicket

The running example in this paper is a smart card based electronic ticketing system called *eTicket*. The UML deployment diagram in Fig. 1 shows the system participants, their communication structure as well as the attacker abilities. The

[1] www.isse.uni-augsburg.de/en/projects/reif/secureMDD/

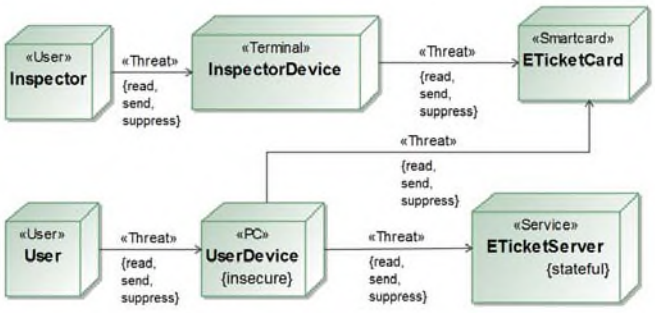

Fig. 1. Deployment diagram of the application

system contains two different types of users. The first type consists of card owners (*User*) who can buy a train ticket online with a *UserDevice* like a PC. This ticket is created on an *ETicketServer* and sent to the user's *ETicketCard*. The other type of system participant is an *Inspector*. Every inspector owns an *InspectorDevice* that is able to communicate with a user's *ETicketCard* and to validate and stamp the tickets stored on an *ETicketCard*. The attacker abilities are modeled with the stereotype ≪Threat≫ with three different properties *(read, send, suppress)* that describe her abilities. The security requirements that should be guaranteed are three application-specific security properties. "Only tickets that were issued by the eTicket server will be stamped by an inspector" (an inspector can differentiate between real and fake tickets), "a ticket will not be lost because of interruptions (e.g. connection errors, an active attacker or a removed smart card) during the buy ticket process" and "a ticket will not be used multiple times". The full model is available on our website.

IV. Modelling of Restrictions

For model checking of large security-critical applications with an attacker who tries to violate the security properties it is common to abstract the formal specification manually to reduce the complexity [2]. But for a model driven approach with automatic generation of the formal specification it is necessary to do the abstraction inside the application model. Furthermore, the original application model must be preserved because it is necessary for the code generation. Therefore, we support the definition of additional test diagrams (described with UML activity and deployment diagrams) which enrich the model with restrictions of protocol runs, attacker abilities and instantiations. From the test diagrams different restricted specifications are generated to test only specific cases.

A. Restriction of protocol runs

A protocol run usually contains several incoming messages that are distributed between several UML activities in SecureMDD. The activities can be combined with other activities to support new functionalities without unnecessary duplication inside the application model. The protocol runs can be restricted in test diagrams by omitting entire activities, defining a specific execution order between the activities or even restrict the number of executions for activities. These restrictions cause a major speed up but require expert know how. As a result it is possible that some attacks are not found.

1) Omit entire activities: Large applications consist of several functionalities that are defined with UML activities. For some security properties only a subset of all activities are relevant. Although in some cases this fact is obvious, it can require a lot of resources to detect automatically which activities are not needed. In the worst case an automatic detection of irrelevant activities and model checking for all activities has the same complexity, because it depends on the security properties, and cause no speed up. In the eTicket example and for the security property "a ticket will not be lost" it is useful to omit some activities manually. Therefore, a test diagram has to be created that contains only the relevant activities (see Fig. 2). This test diagram is transformed to a formal specification that removes the irrelevant activities (e.g. stamp and show tickets). Creating and altering this diagram is very simple in contrast to changing the generated specification.

Fig. 2. Activities for buying a ticket

2) Restrict protocol order: Another way to reduce the complexity and speed up model checking is to restrict the protocol order. Fig. 3 shows six independent activities with a variety of possible combinations which are restricted into one concrete protocol run if the attacker does not interfere. This restriction reduces the possible executions significantly and provides a major speed up. Therefore, it is useful to check predetermined execution orders. To violate the security property "a ticket will not be used multiple times" it is necessary to buy a ticket and to stamp this ticket. A model checker doesn't know which combinations are more useful than others. With our approach the necessary meta information can be expressed inside the application model.

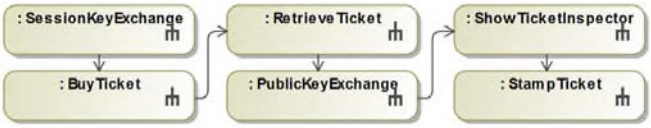

Fig. 3. Buy and stamp a ticket

3) Limit the number of executions: How many resources a specification needs to be model checked depends particularly on how many times an activity can be executed (i.e., how many times the messages inside an activity can be received). There are two ways to describe the possible executions per activity. An activity can be executed at least as often as the activity occurs in a test diagram. Additionally, an execution limit of each activity can be determined by a stereotype called ≪MaxInvocationCount≫ that is applied on a test diagram.

This gives enough variability to determine the possible executions. It makes the definition of restricted executions easy.

B. Restriction of attacker abilities

Beside the mentioned restrictions, the attacker abilities are also a good point to restrict the complexity for model checking of security-critical applications. One way to restrict the attacker abilities is to alter the deployment diagram. Because the attacker abilities are modeled on each connection (see Fig. 1) there are many possible combinations. Each modeled security-critical applications contains a deployment diagram. For each test diagram those attacker abilities can be changed by modeling a new deployment diagram inside the package of the test diagram. Each package contains only one deployment diagram and each test diagram can have other attacker abilities by defining different packages. In some cases this restriction can still be too coarse. Hence, the attacker abilities can also be defined on activities (an activity includes several transmitted messages). That means the attacker abilities for a message are the attacker abilities applied on the activity that contains the message. If no abilities are applied, the attacker abilities from the deployment diagram in the same package is used. If there is no deployment diagram in the package, the main deployment diagram is used. This method gives enough flexibility to define attacker abilities but does not force the modeler to do so for each message.

Fig. 4 describes that a ticket is bought, stored on a smart card, stamped and then the process is repeated once. If a deployment diagram for the test diagram specifies an attacker who can only suppress messages, then Fig. 4 would cause additionally that the attacker is able to read messages that are exchanged inside the first *BuyTicket* activity and to send messages inside the second *BuyTicket* activity. This is a fine-grained approach to restrict and relax attacker abilities. It supports also different abilities on same activities at different times. This is similar to the definition of concrete test cases but introduces a degree of freedom.

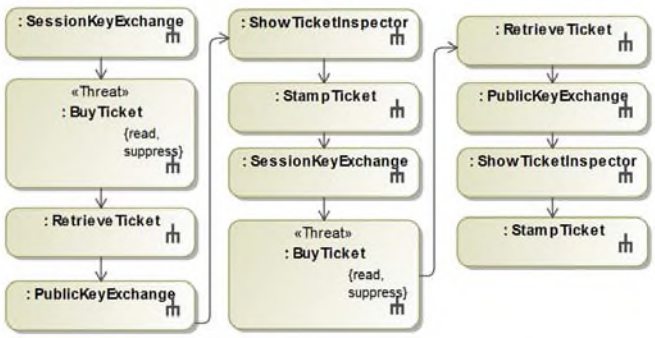

Fig. 4. Restricted attacker abilities and protocol order

C. Restriction of instantiation

Another possibility to reduce the complexity of model checking is to determine the number of instantiations for each participant. The default is that each participant can be instantiated only once. But with that restriction some attacks can not be found. To show for the eTicket example that the same ticket can not be duplicated and stored on different valid smart cards it would be necessary to consider at least two instantiations of *ETicketCard*. Therefore, each participant inside the deployment diagram has the property *MaxInstances* that can change the default setting.

V. ASLan++ Specifications

Our previous work [7] shows the automatic transformation from a SecureMDD application model to an ASLan++[17] specification that can be used for model checking with AVANTSSAR[1]. The example in Sect. III leads to an ASLan specification with 162 transition rules (ASLan++ specifications are translated automatically into ASLan specifications). This is a lot compared to the average 20 transition rules of the AVANTSSAR examples. Because of that, even simple security vulnerabilities could not be found within weeks. To reduce the complexity, several domain-specific automatic abstraction rules were defined. In this way, the 162 transition rules could be reduced to 65 transition rules without any security relevant changes. Thus, the model checker was able to find simple attacks but more complex attacks could not be found even after several weeks model checking with full CPU load on a 3GHz quad core computer. The solution was to reduce the complexity by manually restricting the protocol runs.

In the previous Section we described how to express such manual restriction inside the application model. Textual specifications of large applications can be confusing and manual changes are time-consuming and error-prone. A big advantage of a model driven approach is that an application has different abstraction views. It is much easier to restrict a large application on an abstracted view. A single change on the top level (abstract model) results in many changes on the bottom level (ASLan++ specification). For example, Fig. 2 reduces the size of the original specification by 250 non-contiguous lines and Fig. 4 increases the original specification by 200 lines of code and causes a major speed up for model checking because the attacker abilities are restricted in a fine-grained manner and only specific activities are executed multiple times and only within a restricted order. In Fig. 2 each message can be received independently from the other message. In contrast, in Fig. 3 and 4 all message depend on each other and describe a restricted order that reduces the possible combinations. The generated specifications are available on our website[2].

VI. Automatic vs Manual Abstractions

Automatic abstractions are great but have certain limitations. Manual abstractions need expert know-how and can change the application behavior but can reduce the complexity of each application so that it can be model checked in reasonable time. Our approach provides a framework that makes the creation and management of manual abstractions for model checking

[2]www.isse.uni-augsburg.de/en/projects/reif/secureMDD/

of security-critical applications easy. In our last paper we model checked three application-specific security properties for the electronic ticketing example described in Sect. III. Two security flaws could not be found without manual abstractions even after a week. Despite some minor protocol changes in contrast to our last paper the attacks could still not be found without manual abstractions even after a week.

By omitting some protocol steps like in Fig. 2 the first security flaw could be found in few seconds. The attacker abilities did not have to be restricted. The second security flaw was found in 28 minutes after restricting the protocol order (see Fig. 4) and the attacker abilities. The assumption was that something was wrong within the process for buying and retrieving tickets. Hence, the attacker got the abilities to read, send and suppress all messages inside the *BuyTicket* and *RetrieveTicket* activities. The modeled restrictions are relatively obvious but would be difficult to determine automatically. With the mentioned manual abstractions the three existing security flaws of the eTicket example could be found within some minutes without specifying the entire attack.

VII. Related Work

There are some model driven approaches [9], [5], [3] that use model checking to find security flaws in their modeled applications. But these approaches do not model the full application behavior and do not support application-specific properties. Hence, they have much smaller models for the same applications but the generated code has to be extended by logic that is usually also security-critical. UMLsec [13] describes the modeling of system behavior and application-specific properties but only for a very small example without serious complexity problems with model checking. Model checking of large security-critical applications with fully specified behavior and a strong attacker model can always lead to an explosion of the search space. Therefore, additional models that reduce the complexity for model checking are necessary. [2] considers an abstracted security-critical application but the abstractions are done manually on the textual specification. In [12] an approach is described for selecting a subset of possible test cases to fit resource and time constraints, but it does not consider security.

VIII. Conclusion

Despite the improvement of model checking techniques and the increase of computations speed, model checking will probably still have a problem with complexity of some applications. This depends on the abstraction level and the security properties. In this paper we described a model-driven approach that enables the creation of one application model that contains different abstraction views to generate executable code automatically and model check application-specific security properties for some modeled abstractions. Our approach has the advantage that everything is generated from one model with one modeling language that has the aim to abstract from unnecessary details and make the development of secure applications as easy and clear as possible. We have shown how to restrict protocol runs, attacker abilities and instances inside a UML model that also describes the entire behavior to generate executable code that has not to be extended. We have depicted the specification generated from the modeled restrictions and compared the results between the automatic and manual abstractions. Our result is that model checking needs manual abstractions to get useful results in an acceptable time for real and large applications. With our approach it is very easy to create and change such abstractions without changing the documents that describe the entire application.

References

[1] A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, et al. The avantssar platform for the automated validation of trust and security of service-oriented architectures. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 267–282, 2012.

[2] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 1–10. ACM, 2008.

[3] W. Arsac, L. Compagna, G. Pellegrino, and S. Ponta. Security validation of business processes via model-checking. *Engineering Secure Software and Systems*, pages 29–42, 2011.

[4] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[5] D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security: From UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, pages 39–91, 2006.

[6] M. Borek, N. Moebius, K. Stenzel, and W. Reif. Model-driven development of secure service applications. In *Software Engineering Workshop (SEW), 2012 35th Annual IEEE*, pages 62–71. IEEE, 2012.

[7] M. Borek, N. Moebius, K. Stenzel, and W. Reif. Model checking of security-critical applications in a model driven approach. In *Software Engineering and Formal Methods*. Springer Berlin Heidelberg, 2013.

[8] M. Borek, N. Moebius, K. Stenzel, and W. Reif. Security requirements formalized with OCL in a model-driven approach. In *Model-Driven Requirements Engineering Workshop (MoDRE), 2013 IEEE*. IEEE, 2013.

[9] M. Deubler, J. Grünbauer, J. Jürjens, and G. Wimmel. Sound development of secure service-based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 115–124. ACM, 2004.

[10] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. Kiv: overview and verifythis competition. *International Journal on Software Tools for Technology Transfer*, pages 1–18, 2014.

[11] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. How secure is textsecure? Cryptology ePrint Archive, Report 2014/904, 2014. http://eprint.iacr.org/.

[12] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.*, 22(1):6:1–6:42, Mar. 2013.

[13] J. Jürjens. Model-based security engineering with uml. In *Foundations of Security Analysis and Design III*, pages 42–77. Springer, 2005.

[14] K. Katkalov, N. Moebius, K. Stenzel, M. Borek, and W. Reif. Modeling test cases for security protocols with SecureMDD. *Computer Networks*, (0):–, 2013.

[15] N. Moebius, K. Stenzel, and W. Reif. Modeling Security-Critical Applications with UML in the SecureMDD Approach. *International Journal On Advances in Software*, 1(1), 2008.

[16] NIST. triple handshake attack (cve-2014-1295). Online, 4 2014. Available at http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1295.

[17] D. Von Oheimb and S. Mödersheim. Aslan++ a formal security specification language for distributed systems. In *Formal Methods for Components and Objects*, pages 1–22. Springer, 2012.