CHALMERS
UNIVERSITY OF TECHNOLOGY

SecArchUnit: Extending ArchUnit to support security architectural conformance testing

Master's Thesis in Software Engineering

MARCUS RANDEVIK
PATRIK OLSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020:NN

SecArchUnit: Extending ArchUnit to support security architectural conformance testing

MARCUS RANDEVIK
PATRIK OLSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

SecArchUnit: Extending ArchUnit to support security architectural conformance testing
MARCUS RANDEVIK
PATRIK OLSON

Supervisor: Riccardo Scandariato, Computer Science and Engineering
Examiner: Michel Chaudron, Computer Science and Engineering

Master's Thesis 2020:NN
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

An Informative Headline describing the Content of the Report
A Subtitle that can be Very Much Longer if Necessary
MARCUS RANDEVIK
PATRIK OLSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Keywords: lorem, ipsum, dolor, sit, amet, consectetur, adipisicing, elit, sed, do.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Name Familyname, Gothenburg, Month Year

Contents

List of Figures

List of Tables

1

Introduction

In the age of an ever more digitalized world, ensuring the security of software systems becomes an increasingly more critical task. The increased severeness is true in particular, as systems are being developed based on the principle of permanent connectivity [1]. Services ranging from those offered by governmental agencies to that of social media are always connected to the open internet, potentially creating a large surface of attack. Although the great importance of a securely designed system is widely known, developing a secure system is a challenging task as far from all software engineers are security specialists, or for that matter, particularly educated in security aspects [2].

Introducing a weakness into a system can be done in every part of the software development lifecycle. An early introduction often leads to a more costly fix as the scope of the flaw is increased [3]. Typically, weaknesses are categorized as either implementation bugs (e.g. buffer overflow) or design flaws (e.g. client-side validation) where the later commonly has a more extensive scope and is the responsibility of the architect [4]. However, this binary division might be too simplified as it assumes a securely designed is correctly implemented, meaning that there is no discrepancy between the "intended architecture" and the actual one found in the source code. Jasser [5] considers violations of the architectural security design as a third type of category with a comparable impact to that of design flaws.

Deviations from the intended architecture are not unique to the initial implementation of a system. Over time, subsequent changes made to the system, often due to new requirements, frequently lead to further disparency called software erosion [6]. Proponents of the agile methodology also somewhat worsen the problem of erosion as the reduction of upfront design severely hinders the design of the architecture [7]. Many of the tools and techniques produced by academia to try and remedy the frequent violations of architectural design have failed to gain much adoption in the industry [6]. Of the few that have, architectural design documentation is the most prominent and widely included in numerous software processes. While the technique is well adopted, performing it formally to allow for automatic compliance monitoring is seldomly done, which causes developers to rely on the less scalable method of manual reviews [6].

A recently developed tool called ArchUnit[1] has provided architects with the capability to express architectural constraints trough Java unit testing frameworks. ArchUnit is not the first tool developed for architectural conformance checking. It possibly is, however, the first to leverage already existing testing infrastructure (Unit testing) in a manner that allows for testing over time. In its current form, ArchUnit has not provided any explicit functionality to test security constraints. Thus, the objective of this thesis is to explore the feasibility of expressing and enforcing typical security architectural constraints trough ArchUnit.

1.1 Scope

This study focuses on architectural measures that aid in fulfilling a security goal. Moreover, these measures should be possible to enforce through static analysis of Java bytecode. Any measures related to the configuration (of an application or operating system), the file-system, or other run-time properties that cannot be validated through static code analysis are deemed to be out of scope.

1.2 Research questions

To fulfill the objective of the study, the following research questions have been defined:

- RQ1: What architectural security constraints can be validated using the tool?
- RQ2: What modifications can be made to the tool in order to facilitate the validation of additional constraints?

1.3 Research contribution

This study aims to show how architectural security constraints can be validated with the help of a static analysis tool. The thesis demonstrates applications of the tool to several open source systems, over time, in an evaluation of its efficacy and precision in terms of detecting violations of constraints.

1.4 Limitations

The thesis, and the modified version of ArchUnit, is not aimed at being a complete solution for all security architectural constraints. Instead, the study is performed to provide an initial evaluation of the possibility of using ArchUnit as an alternative to already existing techniques of static conformance checking.

The principle of ArchUnit may very well apply to programming languages other than Java. Though, the limited scope of the thesis makes it unfeasible to provide

[1] https://www.archunit.org/

functionality to analyze source-code in additional languages.

1.5 Thesis outline

The remainder of this report is structured as follows. Outline chapters

Chapter 2: Provides a general background to the topic of software architecture and architectural security constraints. In addition, the ArchUnit framework is introduced and compared to previously developed tools for architectural conformance.

Chapter 3: design of our evaluation, how systems are chosen, the protocol for testing the tool, how to assert the ground truth, process of mapping constraints to rules

Chapter 4: composing the architectural constraints, our final constraints, what they prevent and how they are expressed

Chapter 5: expressing and enforcing constraints with the tool, the identification of missing information / tool features

Chapter 6: results from the empirical validation

Chapter 6: discussion, answer our research questions

Chapter 7: conclusion

2

Background

This chapter...

2.1 Software architecture

[8], [9], [10], [11], [12],

2.2 Architectural security constraints

[13], [1], [14],

Security goals, CIAA...

2.3 ArchUnit

,

2.4 Architectural conformance monitoring

[15], [16], [17], [18], [6], [19], [20], [21],

3

Methodology

This chapter describes the adopted method for collecting relevant constraints and mapping these to the common security-goals of CIAA. Second, this chapter presents the validation plan for expressing security constraints with ArchUnit (as is) by means of an illustration and for expressing additional constraints by means of a controlled experiment.

3.1 Data collection

In order to compose appropriate security architectural constraints, a review of security measures and common weaknesses has been performed. The sources used in this review are presented below.

CAWE catalog: The Common Architectural Weakness Enumeration catalog [22] details 224 common weaknesses in security architectures. Each entry has a description of the weakness and exemplifications of how it could manifest itself in the source code, when applicable. In some entries, there are recommendations on what techniques can be used to detect the weakness, along with mitigation strategies.

Security patterns: Similar to the usage of general design patterns made famous in [23], security patterns provide a reusable and domain-independent solution to a known problem. More specifically, this study focused on security patterns for the design phase, as defined in [2]. While the security pattern repository[1] lists over 170 security patterns, not all are provided with sufficient detail or at the appropriate level of abstraction. As a result, the report by Scandariato et al. [24] which provides a filtered list of patterns.

Security rules: Architectural security rules constrain the implementation of a system while less solution-oriented compared to security patterns. Eden and Kazman differentiate architectural security rules from those defined on a level of source code based on two criteria, locality and intension/extension [25]. Architectural rules are both non-local and intensional, meaning that they affect all or several parts of the system while having "infinitely-many possible instances". In [5], Jasser presents a catalog of architectural security rules. Although the entire catalog of 150 security

[1] http://sefm.cs.utsa.edu/repository/

rules is not yet available, the initial list of 22 included in the paper was used in our study.

3.1.1 Processing

Security measures that we deem difficult to enforce through static analysis, or are otherwise unrelated to security architecture, are discarded upfront. Applicable security measures are categorized according to the security goal that they aim to fulfill. Measures that are similar in nature are combined and presented as a single measure. Additionally, measures defined as the absence of certain functionality are deemed as less important due to the increased difficulty of enforcement [14].

3.2 Validation

The validation of our results will be performed in two ways, depending on whether or not a constraint required any modification of ArchUnit or additional information in the source-code. The latter category carries a higher degree of scientific value thus motivating a more thorough validation procedure. In the sections below, both types of procedures will be described.

3.2.1 Solution Proposal

As described in section 2.3, ArchUnit already provides functionality to perform conformance testing of architectural constraints. However, it is unclear whether the framework supports the enforcement of security architectural constraints. The mapping of security architectural constraints to that of rules in ArchUnit allows us to perform a *proposal of solution* as described in [26]. This type of validation is intended to propose a novel or significantly improved technique without rigorous validation. Instead, a proof of concept or small example is used to facilitate later validation.

3.2.2 Controlled Experiment

In contrast to constraints which may be implemented through already existing functionality, those requiring extension to either the API or additional information within the source-code are not guaranteed to reliably detect violations of the intended architecture. Therefore, validation is performed using a laboratory experiment [27] in order to increase the precision of the measurement.

Integration with already existing testing frameworks is a prominent advantage of ArchUnit. As testing is generally performed overtime to ensure that a system does not degrade, the focus of the experiment is to determine whether ArchUnit can detect changes to security architectural constraints between two versions.

3.2.2.1 Performance metrics

The imbalance between the designer, who needs to ensure that every single aspect of a system is secure, and the attacker, who needs to succeed only once, influences the metrics chosen to represent how well the extension to ArchUnit performs. Precision and recall were the metrics of choice, with the greater importance placed on the latter.

3.2.2.2 Projects used in evaluation

Systems to be included in the validation needed to fulfill two mandatory criteria. First and foremost was the fact that they need open source and written in Java as the static analysis of ArchUnit relies on the source-code. Secondly, there needed to be at minimum two different snapshots in order to fulfill the goal of comparing subsequent changes to a system.

In addition to the mandatory criteria, other aspects were also considered to reduce bias in the validation process. Systems which had already been analyzed in previous literature [28, 18] would provide an existing architecture and its security analysis, which we leverage as ground truth in our experiment. Additionally, systems which have a well documented architecture and security requirements and were within a reasonable size can further help mitigate potential internal validity threats.

4

Selection of Architectural Security Constraints

This chapter describes the methodology of collecting a final list of security constraints from three sources.

4.1 Collection of Security Measures

The processed collection of security measures can be seen in Table 4.1.

4.2 Final Selection

Out of the larger collection of security measures, we have selected 7 entries from diverse security goals and formulated them as architectural security constraints. The selected constraints can be seen in Table 4.2.

#	Measure	Goal	Sub-goal
1	Exceptions should not reveal sensitive information to the client	Confidentiality	Reveal
2	Do not allow sensitive information to bleed to other components	Confidentiality	Reveal
3	Do not send or receive sensitive information in plain text	Confidentiality	Transmit
4	Every outbound message is sent from a central point of the system	Confidentiality	Transmit
5	Strip and validate data before it passes a trust boundary	Confidentiality	Store
6	Do not log secrets	Confidentiality	Store
7	The system must not provide functionality to decrypt secured log messages	Confidentiality	Store
8	Ensure erroneous output is not propagated between components	Integrity	Fault tolerance
9	Input from a user is validated	Integrity	-
10	The user never accesses the session object	Integrity	-
11	Checkpointed system	Availability	Recoverability
12	Limit and/or throttle allocation of resources	Availability	Resource usage
13	Provide a single point of access to the system	Accountability	-
14	There is at least one checkpoint initialized for both tasks: authentication and authorization	Accountability	-
15	Log all security relevant events with appropriate information	Accountability	Auditing
16	Enforce authentication at a central point of the system	Accountability	Authentication
17	Enforce authorization at a central point of the system	Accountability	Authorization
18	Every security-relevant log message must be secured	Accountability	Non-repudiation

Table 4.1: Security measures and their related goals.

#	**Constraint**
1	Log all security events with relevant information
2	Enforce authentication/ authorization at a single point in the system
3	Every outbound message is sent from a central point of the system
4	Input from a user is validated
5	Allocation of resources is limited or throttled
6	Do not allow sensitive information to bleed to other components
7	Do not log secrets

Table 4.2: List of constraints that have been selected for enforcement.

5

Enforcing Constraints

implementation of the constraints is very much a work in progress

This chapter explains how the constraints can be expressed and validated with the tool. The constraints are divided into three distinct categories. The first category contains the constraints that are possible to express in ArchUnit as-is. The second category describes constraints that are enforceable with the help of additional information in source code. The third and final category details constraints that require an extension of ArchUnit to be possible to enforce.

5.1 Support in ArchUnit as-is

ArchUnit contains an extensive vocabulary for expressing typical architectural constraints. These constraints are typically composed of three parts. The first part indicates the type of Java construct that should be inspected. These constructs include classes, methods, fields and constructors. The second part contains a predicate that selects a subset of these constructs. The third part defines the condition that must hold true for all the selected constructs.

An example of a rule defined solely using this standard vocabulary can be seen in Listing 5.1, where each of the three aforementioned parts of the constraint has been separated into their own line. The rule is a simple example of complete mediation, where some internal classes must only be accessed through a mediator.

```
1 ArchRule rule = classes()
2     .that().resideInAPackage("..internal..")
3     .should().onlyBeAccessed().byAnyPackage("..mediator..");
```

Listing 5.1: Example of a rule that is expressed with the standard vocabulary.

In cases where this vocabulary is not sufficient for expressing a constraint, there is a possibility to define custom predicates and conditions over any given construct and supplying these as arguments to the `that()` and `should()` methods.

5.1.1 Constraints

Log all security events with relevant information

This constraint is expressed with the assumption that there is a central class through which security related actions are performed. Any publicly accessible method in

this class must contain at least one call to the logging facility. Both the class that represents the secure service facade and the class responsible for logging are expressed in the constraint itself, meaning there is no need for additional information in the source code.

Enforce authentication/ authorization at a single point in the system

...

Every outbound message is sent from a central point of the system

...

5.2 With Additional Information in Code

Some of the architectural constraints require that the developer injects additional information into the source code.

In some cases, this information is simply an indicator that says something about an entire class. Naming the class with a specific suffix is one approach to accomplish this. Another approach is to implement an empty interface, which is the technique used with Java's `Serializable`[1] interface.

In other cases, however, the information may be required for methods of arbitrary signatures and even specific fields. For the purposes of flexibility and minimal obtrusiveness, any extra information is expressed in the form of annotations. These can be applied to classes, fields, methods and parameters without changing the underlying architecture of the system.

5.2.1 Constraints

Input from a user is validated

...

Allocation of resources is limited or throttled

While resources is a broad term, this constraint focuses on preventing the exhaustion of CPU and memory resources through the creation of new threads and processes. As such, every block of code that contains a call to the `start()` method of a `Thread`[2] or any of its subclasses, must be marked as containing a resource restriction mechanism. The same rule is applied for calls to `ProcessBuilder.start()`[3] and `Runtime.exec()`[3], which lead to the creation of new processes.

[1] https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html
[2] https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
[3] https://docs.oracle.com/javase/7/docs/api/java/lang/Process.html

The marking is done with the help of an annotation, either on the relevant method or the entire class. The decision of how the restriction mechanism is implemented is left to the developer of the system.

5.3 With Extensions to ArchUnit

In the current ArchUnit API, a rule that aims to constrain access to a method (or field) must be expressed in terms of the type signatures of the source and target methods. Some of our constraints require knowledge about the type signature of the object that is being passed as a parameter. This is a non-issue when fields and method parameters are of the same types as the objects being passed to them. However, in cases where a method signature accepts a "more general" type, such as an `Object`, there is no way for ArchUnit to constrain the types of the objects that are actually being passed as parameters.

ArchUnit builds its representation of the architecture using ASM[4], a Java bytecode analysis framework. This framework contains functionality for keeping track of the stack and local variables while analyzing the instructions of a method. With knowledge of the type signatures of the references on the stack at the time of a method call or field assignment, it is possible to determine the type signatures of objects passed as parameters or an object being assigned to a field. Our extension provides this additional information in ArchUnit's representation of accesses to fields and methods, which the rule definitions can then make use of.

5.3.1 Constraints

Do not allow sensitive information to bleed to other components

...

Do not log secrets

...

[4]https://asm.ow2.io/

6

Evaluation

adjust sections as necessary

6.1 Strategy

6.2 Selection of systems

Bibliography

[1] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security Testing," in *Advances in Computers*. Elsevier, 2016, vol. 101, pp. 1–51. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0065245815000649

[2] N. Yoshioka, H. Washizaki, and K. Maruyama, "A survey on security patterns," *Progress in Informatics*, no. 5, p. 35, Mar. 2008. [Online]. Available: http://www.nii.ac.jp/pi/n5/5_35.html

[3] G. Mcgraw, "Software security," *IEEE Security & Privacy Magazine*, vol. 2, no. 2, pp. 80–83, Mar. 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1281254/

[4] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield, and others, "Avoiding the top 10 software security design flaws," *IEEE Computer Society*, 2014.

[5] S. Jasser, "Constraining the Implementation Through Architectural Security Rules: An Expert Study," in *Product-Focused Software Process Improvement*, X. Franch, T. Männistö, and S. Martínez-Fernández, Eds. Cham: Springer International Publishing, 2019, vol. 11915, pp. 203–219. [Online]. Available: http://link.springer.com/10.1007/978-3-030-35333-9_15

[6] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, Jan. 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121211002044

[7] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105–119, Mar. 2002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121201001522

[8] D. Garlan and M. Shaw, "AN INTRODUCTION TO SOFTWARE ARCHITECTURE," in *Series on Software Engineering and Knowledge Engineering*. WORLD SCIENTIFIC, Dec. 1993, vol. 2, pp. 1–39. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/9789812798039_0001

[9] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. Pittsburgh, PA, USA: IEEE, 2005, pp. 109–120. [Online]. Available: http://ieeexplore.ieee.org/document/1620096/

[10] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 3rd ed., ser. SEI series in software engineering. Upper Saddle River, NJ: Addison-Wesley, 2013.

[11] R. Scandariato, K. Yskout, T. Heyman, and W. Joosen, "Architecting software with security patterns," Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW 515, Apr. 2009.

[12] P. H. Nguyen, K. Yskout, T. Heyman, J. Klein, R. Scandariato, and Y. Le Traon, "SoSPa: A system of Security design Patterns for systematically engineering secure systems," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ottawa, ON, Canada: IEEE, Sep. 2015, pp. 246–255. [Online]. Available: http://ieeexplore.ieee.org/document/7338255/

[13] M. Broy, J. Grünbauer, and C. A. R. Hoare, Eds., *Software system reliability and security*, ser. NATO security through science series. Amsterdam ; Washington, DC: IOS Press, 2007, no. v. 9, oCLC: ocn127107624.

[14] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, "Security Requirements Engineering: A Framework for Representation and Analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, Jan. 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4359475/

[15] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* Orlando, FL, USA: ACM, 2002, pp. 187–197. [Online]. Available: http://ieeexplore.ieee.org/document/1007967/

[16] M. Abi-Antoun and J. M. Barnes, "Analyzing Security Architectures," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 3–12, event-place: Antwerp, Belgium. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859001

[17] D. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, Sep. 1995. [Online]. Available: http://ieeexplore.ieee.org/document/464548/

[18] M. Abi-Antoun and J. Aldrich, "Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations," in *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA 09.* Orlando,

Florida, USA: ACM Press, 2009, p. 321. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1640089.1640113

[19] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*. Mumbai, India: IEEE, Jan. 2007, pp. 12–12. [Online]. Available: http://ieeexplore.ieee.org/document/4077029/

[20] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting after the fact: Recovering architectural design decisions," *Journal of Systems and Software*, vol. 81, no. 4, pp. 536–557, Apr. 2008. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016412120700194X

[21] Hong Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: a system for discovering architectures from running systems," in *Proceedings. 26th International Conference on Software Engineering.* Edinburgh, UK: IEEE Comput. Soc, 2004, pp. 470–479. [Online]. Available: http://ieeexplore.ieee.org/document/1317469/

[22] J. C. S. Santos, K. Tarrit, and M. Mirakhorli, "A Catalog of Security Architecture Weaknesses," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Gothenburg, Sweden: IEEE, Apr. 2017, pp. 220–223. [Online]. Available: http://ieeexplore.ieee.org/document/7958491/

[23] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*, ser. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995.

[24] R. Scandariato, K. Yskout, T. Heyman, and W. Joosen, "A system of security patterns," Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW 469, Dec. 2006.

[25] A. Eden and R. Kazman, "Architecture, design, implementation," in *25th International Conference on Software Engineering, 2003. Proceedings.* Portland, OR, USA: IEEE, 2003, pp. 149–159. [Online]. Available: http://ieeexplore.ieee.org/document/1201196/

[26] R. Wieringa, N. Maiden, N. Mead, and C. Rolland, "Requirements engineering paper classification and evaluation criteria: a proposal and a discussion," *Requirements Engineering*, vol. 11, no. 1, pp. 102–107, Mar. 2006. [Online]. Available: http://link.springer.com/10.1007/s00766-005-0021-6

[27] K.-J. Stol and B. Fitzgerald, "The ABC of Software Engineering Research," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 1–51, Sep. 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3276753.3241743

[28] S. Peldszus, K. Tuma, D. Struber, J. Jurjens, and R. Scandariato, "Secure Data-Flow Compliance Checks between Models and Code Based on

Automated Mappings," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Munich, Germany: IEEE, Sep. 2019, pp. 23–33. [Online]. Available: https://ieeexplore.ieee.org/document/8906984/