Model-Based Assessment of Conformance to Acknowledged Security-Related Software Architecture Good Practices

Monica Buitrago[1], Isabelle Borne[1] and Jérémy Buisson[2]
[1]*IRISA, Université de Bretagne Sud, France*
[2]*CRéA, École de l'Air et de l'Espace, France*

Keywords: Software Architecture, Security-by-Design, Metric, Security, Model-Based Engineering.

Abstract: Security-by-design considers security throughout the whole development lifecycle, to detect and fix potential issues as early as possible. With this approach, the software architect should assess some security level of the software architecture, to predict whether the software under development will have security issues. Previous works proposed several metrics to measure the attack surface, the attackability, and the satisfaction of security requirements on the software architecture. However, proving the correlation between these metrics and security is far from trivial. To circumvent this difficulty, we propose new metrics rooted in CWE, NIST guidelines and security patterns. So, our four novel metrics measure the conformance of the software architecture to these acknowledged security-related recommendations. The usage of our metrics is evaluated with case studies.

1 INTRODUCTION

Security by design has emerged as addressing the security concerns at every stage of software development (Waidner et al., 2014). In this regard, Common Weakness Enumeration (CWE), controls enumeration (NIST, 2020), and security patterns (Fernandez-Buglioni, 2013) provide acknowledged effect of design choices on the security of the software under development. Though, their practical usability were questioned (Yskout et al., 2015). Besides, various metrics have been proposed to quantify, when designing the software architecture, and the attack surface (Alshammari et al., 2009; Gennari and Garlan, 2012; Manadhata and Wing, 2011), the attackability (Skandylas et al., 2022). But, correlating between these metrics and the overall security of the software is challenging as software security is unobservable (Herley and van Oorschot, 2018).

To avoid the difficulty of relating metrics to security, we propose to *measure the extent to which a design,* as reflected in the software architecture*, conforms to the recommendations* cited above. Because these recommendations are acknowledged to reduce the number of vulnerabilities, our metrics are linked to the guarantee of security.

Section 2 summarizes the state of the art, focused on measuring security during software design and development. Section 3 outlines how we position our work in the software engineering process. Section 4 describes our metrics. Section 5 discusses the validity of the metrics on case studies based on real-world applications. Section 6 gives the final remarks and future directions for our work.

2 RELATED WORK

Software quality attributes are stated or implied, non-functional requirements, such as *maintainability*, *performance*, *security*, and so on. Metrics are provided to quantify how well such requirements are satisfied. That way, the intrinsic subjectivity in non-functional assessment is confined to the choice of the metrics (Sommerville, 2016). Measuring on the architecture description enables the prediction of the non-functional assessment for the future software as soon as the design phase.

2.1 Measure the Attack Surface

The attack surface is the set of software elements that an attacker may exploit. A designer can use annotations to specify the sensitive elements, e.g., in the class diagram (Jürjens, 2002). Assuming that *public* means *exposed*, the designer can compute, e.g., the ratio of public attributes among the sensitive ones, and

Buitrago, M., Borne, I. and Buisson, J.
Model-Based Assessment of Conformance to Acknowledged Security-Related Software Architecture Good Practices.
DOI: 10.5220/0012312400003645

In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pages 117-124
ISBN: 978-989-758-682-8; ISSN: 2184-4348

the ratio of methods using sensitive attributes to quantify the attack surface (Alshammari et al., 2009).

(Manadhata and Wing, 2011) proposed the ratio between the effort needed to attack and the potential damage. Looking at a component, the *potential damage* follows from the privilege level at which the component functions are executed, from the ports protocols, and from the kind of storage used by the components. The *attack effort* is the requirements to access the ports. (Gennari and Garlan, 2012) adapted the metrics by looking at the connectors and the ports, instead of the ports and data flows respectively.

All these metrics relate to some concept of *entry point*, i.e., public attributes and methods (Alshammari et al., 2009), and ports (Manadhata and Wing, 2011; Gennari and Garlan, 2012). All of them compute the proportion of entry points. Because we do not focus on the attack surface but on the conformance to patterns and guidelines, we are led to consider the raw number of entry points.

2.2 Measure Security Properties

According to the ISO/IEC 25010 definition, *security* is refined into: *availability*, *confidentiality* and *integrity*. Metrics can be targeted at these properties.

(Siavvas et al., 2021) proposed to use 11 base metrics to score *confidentiality*, *integrity* and *availability*: complexity, cohesion, coupling, encapsulation, and seven rules from the PMD[1] static code analyzer. Like in (Wagner et al., 2015), the metrics are normalized according to an empirical study, and the weights follow from experts' knowledge. A *security index* is the average of the three scores. The empirical study used to compute the normalization parameters also ensures the correlation with the security properties.

We ensure the link with security by measuring the conformance to security patterns and guidelines.

2.3 Other Security-Related Metrics

In (Du et al., 2019), the functions are ranked according to metrics assumed to be correlated with vulnerabilities: number of parameters, number of pointer arithmetic operations, number of nested control structures, etc. Because these metrics are also correlated to the code complexity and size, a preliminary binning-by-cyclomatic-complexity phase is performed as a kind of normalization of the metrics. Such details about the software are not yet available during the architecture stage of design, at which our work applies.

(Casola et al., 2020) model applications as software services (SaaS) hosted by virtual machines (VM). Cloud service providers (CSP) provide VM and SaaS. The components (SaaS, VM, CSP) and their relations (provide, host, use) describe the architecture. Security service level agreement (sSLA) templates model the requirements and controls (choices, levels, parameters) at components. An overall sSLA is combines the component-level templates according to the architecture, using (Rak, 2017)'s algorithm. On the requirement side, the sSLA templates result from risk analysis. On the security assessment side, they result from a *review by the developers* to measure the implemented controls and the related metrics. The metrics we propose, by measuring how much the architecture conforms to the guidelines, may replace the manual review with an automated tool.

(Skandylas et al., 2022) consider a flat assembly of components and connectors, in which some ports are annotated with vulnerabilities. To reflect the adversary's profile, each vulnerability has a probability of success and a cost. A control has an effectiveness (probability of countering) and a cost. Attackability and defendability are computed using these parameters: the probability that the adversary takes over the system, the minimum/maximum attack cost, and the minimum defender's cost. In our work, we propose how to compute the security level of the software, which is assumed by (Skandylas et al., 2022) and matched with the adversary's profile.

3 APPROACH OVERVIEW

Figure 1 depicts how our work integrates with software and security engineering processes. After a requirement engineer produced the software requirements, including the security requirements derived from a *preliminary* risk assessment, the (software) architect designs the (software) architecture. The architect begins with a coarse-grained one, made of few big components connected by connectors. Then, (s)he refines the architecture by recursively decomposing the components into composite structures until (s)he identifies the primitive components (those which are not further decomposed). The *design* risk assessment activity identifies the risks that result from the design decisions made in the architecture. Taking into account this feedback, the requirement engineer updates the (security) requirements. The architect revises the architecture accordingly. This loop is repeated until the remaining risks are acceptable enough (Sommerville, 2016). The resulting detailed architecture is then passed on to the developers, who use it as the specification of each primitive component. In this paper, we do not describe the other steps.

[1] https://pmd.github.io/ (visited on 12/06/2023)

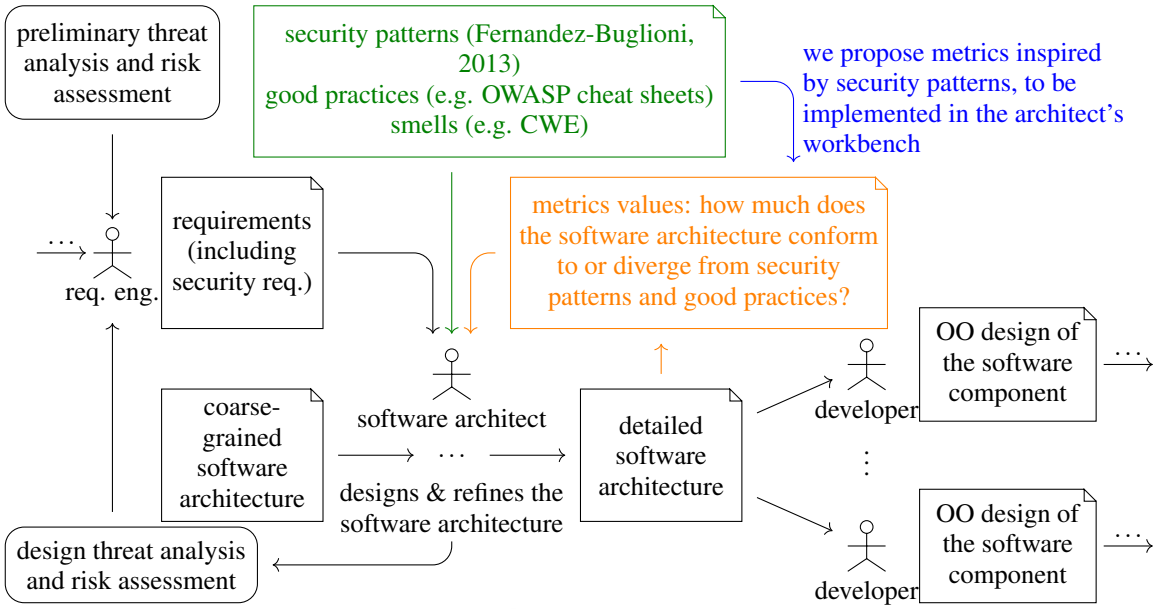

Figure 1: Our work in the software engineering process.

To design the architecture, the architect relies on well-established knowledge such as patterns. There is specific guidance to deal with software security such as (Fernandez-Buglioni, 2013; NIST, 2020).

We propose metrics, inspired by these security-related guidance, to provide the architect with a new security-targeted analysis of the detailed architecture. By integrating this new analysis in her/his workbench, the architect gets feedback about how much the architecture conforms to the guidance (the orange loop in Figure 1). According to the result of this analysis, the architect may decide either to revise the architecture, or to move forward in the engineering process.

Table 1: Supporting guidelines for our metrics.

#ep/c	`SA-8(13)`: *minimized security elements* `SA-8(14)`: *least privilege* `SA-17(7)`: *structure for least privilege* `SC-2`: *separation of system and user functionalities* `CWE-653`: *improper isolation or compartmentalization*
#epp, epd	`SA-8(3)`: *modularity and layering* `SA-8(4)`: *partially ordered dependencies* `CWE-1054`: *invocation of a control element at an unnecessarily deep horizontal layer* `CWE-1092`: *use of same invokable control element in multiple architectural layers*
#lc	`SC-7(13)`: *isolation of security tools, mechanisms, and support components*

guidelines: (NIST, 2020) and https://cwe.mitre.org

4 PROPOSED METRICS

Section 4.1 defines the model of software architecture, and the notations we use. Three of our four metrics are locally defined for one component, which can be chosen from a specific subset. Section 4.2 describes these local metrics. Section 4.3 explains how to consolidate the local measures into architecture-wise values. Section 4.4 describes a fourth metric, which applies intrinsically to the architecture. Table 1 gives the guidelines supporting each metric.

4.1 Architecture Model and Notations

Following the usual approach, e.g. UML, the architecture is made of instances (the *parts*) of *components*. Each component declares *ports* at which *connectors* can be attached to transport the interactions between parts such as method calls, data flow, etc. We distinguish *composite* components (or simply composites), which are assemblies of parts and connectors. The other components, of which the content is omitted from the architecture, are the *primitive* components (or simply primitives). In our work, we use the terms *component*, *connector*, *part*, and *port* with their definition in UML. Our architecture description language is the combination of the UML component and composite structure diagrams.

From the UML architecture description, we extract a view that enforces a strict two-level hierarchy of composites, i.e., the simplified view is composed of

composites, which are themselves composed of primitives (section 4.3 explains why it is not a restriction). We restrict to binary directed connectors, and we allow them to cross the composite boundaries. We ignore ports and types.

In the rest of the paper, we use the notations:

- $\mathcal{A} = \mathcal{P}, \mathcal{E}$ – an architecture, where $\mathcal{P}$ is the set of composites and $\mathcal{E}$ is the set of connectors.
- c, d – some composites ($c \in \mathcal{P}$, $d \in \mathcal{P}$).
- $\mathcal{V}$ – the set of primitives $\mathcal{V} = \bigcup_{c \in \mathcal{P}} c$.
- a, b – some primitives ($a \in \mathcal{V}$, $b \in \mathcal{V}$).
- C – a function that maps a primitive component to its enclosing composite ($\mathcal{V} \mapsto \mathcal{P}$).
- $e = a \to b$ – a connector ($e \in \mathcal{E}$); e is in c if and only if $a \in c \wedge b \in c$.
- $shortest\_path^c(a, b)$ – a function that returns the shortest path from a to b, by following only the connectors in c.

4.2 Local Metrics

Our three local metrics are the number of entry points per composite ($\#ep/c$), which applies to a composite; the number of entry point predecessors ($\#epp$) and the entry point depth (epd), which are computed for primitives at the entry point positions of a composite.

A primitive b is an *entry point* of a composite c if it belongs to c (i.e. $b \in c$), and there is at least one dependent a that does not belong to the same composite (i.e. $a \to b \in \mathcal{E}$ and $a \notin c$).

(NIST, 2020; Fernandez-Buglioni, 2013) emphasize the principle of *process isolation*: a software system must be decomposed into processes, each of them having its own address space and communicating only through well identified ports. Even if, in software engineering, composites are not intended to be deployment domains, execution domains nor security domains, we assume the encapsulation in a composite ensures isolation. Thus, the consequences of process isolation apply to composites. For instance, according to the principle `SA-8(3)` *modularity and layering*, the modularity of the architecture should extend beyond *functional* modularity to the security concerns. Besides, the entry point components are exposed to messages that are not under the control of the composite they belong to. As such, they are responsible to enforce all the security-related functions, including the protection of the communication channel, authentication of the client components, authorization and access control, accountability, audit, input validation and sanitization. When the ports of the composite act as no more than forwarders, the entry points of the composite play the role of the *protected entry points* (Fernandez-Buglioni, 2013).

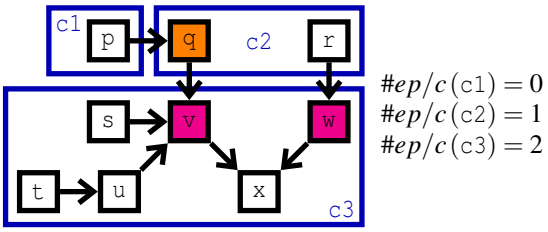

Figure 2: Illustration of the $\#ep/c(c)$ metric.

4.2.1 Number of Entry Points per Composite

On the one hand, the principle `SA-8(13)` *minimized security elements* pinpoints that security-critical components (such as entry points) require specific attention that increases their cost and complexity. So, these components should be as few as possible. This concern matches the reduction of the attack surface by reducing the ratio of publicly exposed ports, functions, methods, attributes (Alshammari et al., 2009; Gennari and Garlan, 2012; Manadhata and Wing, 2011).

On the other hand, the principles `SC-2` *separation of system and user functionalities* and `SA-8(14)` *least privilege*, their consequence `SA-17(7)` on the architecture, and `CWE-653` *improper isolation or compartmentalization* advise to have as many ports as privileges to access the component.

Altogether, these design principles highlight that there is a trade-off between two contradictory goals: on the one hand, having fewer entry points to manage the cost and overhead of secure development by reducing the amount of concerned components; on the other hand, ensuring distinct entry points for each kind of clients and privileges. To help the architect to decide whether this trade-off is satisfied, we define a first metric $\#ep/c(c)$ that simply counts how many entry points a composite component $c \in \mathcal{P}$ contains:

$$\begin{aligned} ep(c) &= \{b \mid b \in c \wedge \exists a, a \to b \in \mathcal{E} \wedge a \notin c\} \\ \#ep/c(c) &= |ep(c)| \end{aligned}$$

$ep(c)$ is the set of the entry points of c. The metric value is the cardinal of this set.

Figure 2 illustrates this metric on a synthetic case:

- The composite component `c1` contains only one primitive component `p`, which has no inbound connector. For this reason, `p` is not considered an entry point of `c1`, and therefore $\#ep/c(\texttt{c1}) = 0$.
- `c3` contains six primitive components (`s`, `t`, `u`, `v`, `w` and `x`). Among them, `v` has three inbound connectors, but only `q` is outside `c3`; `w` has one inbound connector and `r` is outside `c3`; and the inbound connector of `u` and the two inbound connectors of

x come from the inside of c3. So, $\#ep/c(c3) = 2$, the entry points are v and w.

To model the fact that $\#ep/c(c)$ should be neither too small nor too high, we consider that $\#ep/c(c)$ should be in an ideal range $[l_{\#ep/c}, u_{\#ep/c}]$ given by an expert (l the lower bound, u the upper bound of the range). So, we can piecewisely define a distance between the value of the metric and this range, e.g.:

$$d\#ep/c(c) = \begin{cases} |l_{\#ep/c} - \#ep/c(c)| \text{ when } \#ep/c(c) < l_{\#ep/c} \\ 0 \text{ when } \#ep/c(c) \in [l_{\#ep/c}, u_{\#ep/c}] \\ |\#ep/c(c) - u_{\#ep/c}| \text{ when } \#ep/c(c) > u_{\#ep/c} \end{cases}$$

We derive a score from this distance, e.g.:

$$s\#ep/c(c) = (1 + d\#ep/c(c))^{-\alpha} \text{ where } \alpha > 0$$

4.2.2 Number of Entry Point Predecessors

Like explained in the description of $\#ep/c$ (section 4.2.1), the entry points of the composites are some of the trusted components that must implement security concerns, as they are exposed to clients out of the control of the encompassing composite. The second metric $\#epp(b)$ counts how many predecessors a given entry point has in its encompassing composite. If the software architecture is suitably layered, i.e., following the principle SA-8(3) *modularity and layering*, the entry points should not have any such predecessor. When there are some, the developer may misidentify the entry points and (s)he may fail to implement the suitable security concerns. Besides, the principle SA-8(4) *partially ordered dependencies*, the smell CWE-1054 *invocation of a control element at an unnecessarily deep horizontal layer*, and the smell CWE-1092 *use of same invokable control element in multiple architectural layers* defend a strict layering of the architecture.

Let $c \in \mathcal{P}$ be a composite, and let $b \in ep(c)$ be an entry point component of c. The $\#epp(b)$ metric is:

$$epp(b) = \left\{ p \mid \begin{array}{l} C(p) = C(b) \wedge \\ shortest\_path^{C(b)}(p,b) \neq \bot \end{array} \right\}$$
$$\#epp(b) = |epp(b)|$$

Figure 3 illustrates the $\#epp$ metric:

- The predecessor p of q does not belong to the same composite as q. So, it is not counted and $\#epp(q) = 0$. The same applies for w.
- The component v has five direct or indirect predecessors. Among them, only s, t and u are counted because v is reachable only from these three components, when only considering the assembly connectors within c3, the composite of v.

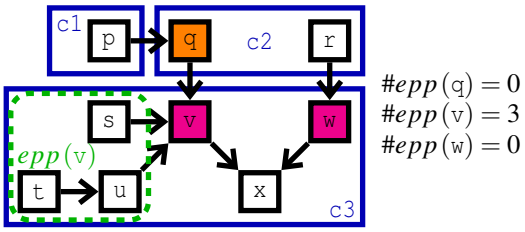

Figure 3: Illustration of the $\#epp(b)$ metric.

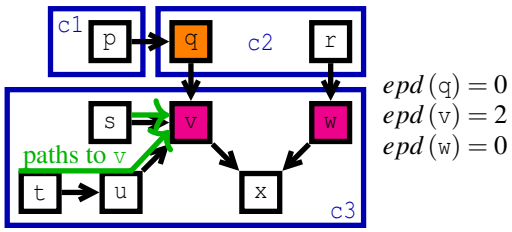

Figure 4: Illustration of the $epd(b)$ metric.

We derive a score from $\#epp(b)$, e.g.:

$$s\#epp(b) = (1 + \#epp(c))^{-\alpha} \text{ where } \alpha > 0$$

4.2.3 Entry Point Depth in Composite

Like stated in section 4.2.2, the CWE and NIST controls recommend the layered architecture. To reflect this, we refine $\#epp$ by considering the depth of the entry point, i.e., the maximum length of the shortest paths from the predecessors to the entry point.

Let $c \in \mathcal{P}$ and $b \in ep(c)$ be an entry point of c. The metric evaluates the shortest path from any predecessor p of b within c to b, and returns the maximum length among these shortest paths:

$$epd(b) = max \left\{ len\left(shortest\_path^{C(b)}(p,b)\right) \mid C(p) = C(b) \right\}$$

In Figure 4, v has three predecessors in its enclosing composite c3. The shortest path from s to v contains one edge; the one from t to v contains two edges; and the shortest path from u to v contains one edge. So, the computed depth of v in c3 is 2.

Similarly to $\#epp$, we derive a score, e.g.:

$$sepd(b) = (1 + epd(c))^{-\alpha} \text{ where } \alpha > 0$$

4.3 From Raw Local Metrics to Architecture-Wise Metrics

As presented in section 4.2, the $\#ep/c$ metric and its derived score are local to a composite component; the $\#epp$ and epd metrics and their derived scores are local to a primitive component (which is expected to

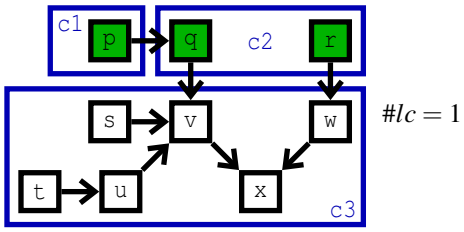

Figure 5: Illustration of the #*lc* metric.

be the entry point of a composite component). The question arises how to lift the metrics and scores up to the enclosing composite, then up to the composite that models the whole software architecture.

The principle `SA-8(9)` *trusted components* (NIST, 2020) advocates that the least trustworthy component in a composite gives the trustworthiness score to the composite. (Casola et al., 2020; Rak, 2017) proposed composition rules based on conjunctions, which boils down to the same principle of the least trustworthy component. (Gennari and Garlan, 2012; Manadhata and Wing, 2011) proposed to sum or average. We choose to use summarizing statistics (minimum, mean and maximum), and we leave the interpretation to the architect.

4.4 Number of Leaf Composites

The principle `SC-7(13)` of *isolation of security tools, mechanisms, and support components* aims at preventing an adversary to gain information on the security tools. Concretely, log collection for audit purpose, security operation centers for supervision, and other security-related functions should use a distinct infrastructure (communication channels, storage) than the one of the application. These infrastructure elements appear in the architecture as additional composites, which do not depend on the ones involved in the functional services. For instance, the architecture may contain one database for the functional services, another database for the log storage, and yet another one for the access control data. The mutual isolation of these storage components appear as each of them being leaves in the architecture. So, the number of leaf composites should not be low.

$$\begin{aligned} lp(c) &= \{a \mid a \in c \wedge \exists b, a \to b \in \mathcal{E} \wedge b \notin c\} \\ lc &= \{c \mid lp(c) = \emptyset\} \\ \#lc &= |lc| \end{aligned}$$

Given $c \in \mathcal{P}$, a composite, its *leave points* $lp(c)$ are its inner components a that depend on some component b outside of c. A composite c is a leaf composite if it has no leave point. The metric value is the cardinal of the set lc of leaf composites.

In Figure 5, `p` is a leave point of `c1`, and `q` and `r` are leave points of `c2`. On the other hand, `c3` has no leave point component. So, the set of leaf composites is $\{$`c3`$\}$. In this architecture $\#lc = 1$.

The score encodes *the higher the better*, e.g.:

$$s\#lc = 1 - (1 + \#lc)^{-\alpha} \text{ where } \alpha > 0$$

5 EVALUATION

We extended the Eclipse Papyrus[2] modeling workbench. Applied to a component in a UML composite structure diagram, our new *compute metrics* command loads the underlying model elements from the XMI files, builds the simplified view like presented in section 4.1, computes the metrics and scores described in section 4.2, 4.3 and 4.4, and reports the results to the software architect.

5.1 A Real-World Application: Xwiki

The Xwiki project[3] is an open-source wiki written in Java and deployed in any compliant Servlet container. To obtain its architecture, we first checked out the 623 kLOC in 8375 Java source files at the `15.4` tag from its GitHub repositories[4] and we compiled it in its bare profile. After reconstructing the UML class diagram (8853 classes and interfaces, and 10634 associations of interest) by using the ASM library[5], we recovered the primitive components in UML component diagrams. In addition to the servlets, we relied on the JSR-330 (`javax.inject`) annotations, along with few Xwiki-specific annotations. Our tool found 2405 primitives. To simplify, we interchanged a component and its instances; or, equivalently, we abusively assumed that there is a single instance of each component. Following the semantic of the component framework, connectors were generated by resolving to any component with a matching name and that provides an interface with a compatible type. In case of ambiguity, our tool selected the component with the most specific interface and implementation class. But, our tool ignored the descriptors provided out of the Java code. It resulted in 8783 connectors. Last, to simplify, we used the 304 generated `jar` artifacts as the composites.

Table 2 gives the score at the level of the architecture. In Table 3, the architect focuses her/his attention on the worst composites according to a selected

[2] https://www.eclipse.org/papyrus/ (visited on 04/07/2023).

[3] http://www.xwiki.org (visited on 04/07/2023).

[4] Repositories `xwiki-commons`, `xwiki-rendering`, and `xwiki-platform` in https://github.com/xwiki.

[5] https://asm.ow2.io/ (visited on 04/07/2023).

Table 2: Scores for the architecture of Xwiki 15.4.

metric	min	avg	max
$s\#ep/c$	0.015	0.64	1.0
$s\#epp$	0.007	0.73	1.0
$sepd$	0.083	0.76	1.0
$s\#lc$	0.83		

Table 3: Worst composites of Xwiki according to $s\#ep/c$.

Composite/Artifact c	$s\#ep/c(c)$	$\#ep/c(c)$
`platform-oldcore`	0.015	70
`platform-model-api`	0.036	31
`commons-extension-api`	0.071	17

score (here $s\#ep/c$). Regarding `platform-oldcore`, the Xwiki developers acknowledge that it should be exploded into smaller ones[6]. Such a change would improve the compartmentalization (`SC-2`, `CWE-653`).

Regarding `platform-model-api`, the high number of entry points results from the fact that it contains several variants of the same services. For instance, it contains 7 implementations of "*entity reference serializer*" that are entry points. Like `platform-oldcore`, it can be split into smaller composites, to avoid that all the variants have equal access to resources, possibly violating the *least privilege* principles `SA-8(14)` and `SA-17(7)`.

Regarding `commons-extension-api`, the architect's investigation shows that each entry point of this composite regards specific concerns (factory, cache, validation, repository, etc.). So, the architect decides that `commons-extension-api` is fine with respect to `SA-17(7)` and other related guidelines, and that the threshold $u_{\#ep/c}$ (see section 4.2.1) was too low for this case. The architect can decide not to investigate additional composites, like for any other analysis (Sommerville, 2016).

Due to space limitation, we do not discuss the other metrics. In summary, we successfully exploited the metrics with a large real-world application.

5.2 Effects of Architecture Variations

Our third experiment aims at observing how our metrics behave in the face of architecture variations, which are not expected to affect the security of the software. To this end, we reverse-engineered the Bitwarden application [7], an open-source distributed password manager. This application is structured into a back-end, a database and several front-end applications (web site, mobile applications and browser plugins). We extracted the class diagrams of the back-end and of the web site (1833 classes and interfaces). Then, by looking at the patterns used in the .Net Core dependency injection framework and its counterpart in Angular TypeScript, we recovered the 144 primitives and most of the connectors. Only the HTTP-based connectors were recovered by hand. Last, we recovered the composites thanks to graph clustering.

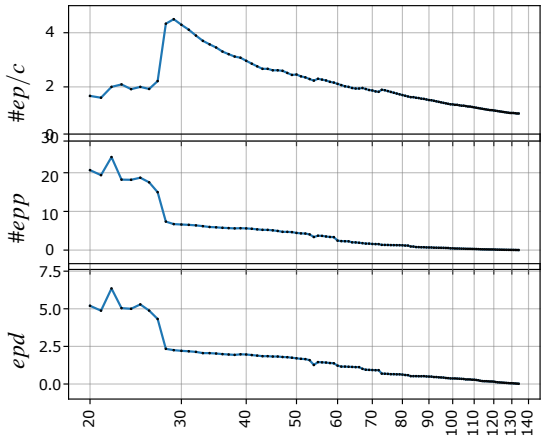

Figure 6: Evolution of the metrics when the number of composites changes.

The graph clustering algorithm we used is parameterized by the number of expected composites: it is the variable in this experiment. Figure 6 shows how the value of the metrics evolves when we change this parameter, averaged over the whole architecture. In the 20-27 range, the clustering algorithm recognizes the web site and the back-end server, even if the boundary is approximate: the unique component implementing the calls to all the back-end APIs (named `ApiService`) is put in the back-end, instead of the web site. The low variations of the metrics in this range are explained by the fact that, in the 20-27 range, the additional composites created by the clustering algorithm contain only few primitives[8]. Such composites have little effect on the averaged metrics nor on the supporting guidelines.

At 28 composites, the clustering algorithm moves `ApiService` from the back-end to the web site (like in the actual source code). This change greatly increases the number of entry points (one entry point per API component, instead of a single one), which appears in the $\#ep/c$ curve in Figure 6. This change effectively improves conformance with, e.g., the principles `SC-2` *separation of system and user functionalities* and `SA-17(7)` *structure for least privilege*.

[6]https://github.com/xwiki/xwiki-platform/blob/xwiki-platform-15.4/xwiki-platform-core/xwiki-platform-oldcore/README (visited on 14/09/2023).

[7]https://www.bitwarden.com/ (visited on 07/07/2023).

[8]The explanation of this behavior of the clustering algorithm is irrelevant according to the topic of this paper.

Beyond 28 composites, the regular decrease is explained by the fact that, as the requested number of composites increases, their size decreases until eventually each composite contains exactly one primitive. Our metrics are irrelevant in such a case, which boils down until you have no composite.

6 CONCLUSION

In this paper, we contribute to address the challenge of providing the software architect with means to evaluate whether an architecture will yield a secure system, without exploitable vulnerabilities. We do so by proposing metrics rooted in acknowledged guidelines. This last point is one novelty of our work in comparison to related works. In the end, it appears that the metrics we propose are different from the ones previously proposed in the related works. Our focus on the patterns, guidelines and smells ensures a direct link to security concerns and intrinsically pinpoints suggestions for improving the architecture, complementing other previously existing metrics.

We used Xwiki, a large open-source application, to ensure that an architect can use our metrics to identify potential security-related weaknesses and improvements in her/his architecture, by referring to the supporting guidelines. Using Bitwarden, another open-source application, we showed that our metrics behave well when the architect modifies the architecture composites.

The main threat to validity is the fact that, in our experiments, we played the role of the architect. We need to setup a controlled experiment with engineers to confirm our results. Besides, our reverse engineering process for Xwiki and Bitwarden is approximate. Still, our observations and conclusions are drawn on the recovered architectures, not on the real applications. Although we believe that, therefore, this limitation of our experiments does not threaten the validity of our conclusions, it does emphasize that our work assumes that the architecture model is available.

In this paper, metrics focus on components, and more specifically composites, which are well suited to study isolation, compartmentalization, and separation of functions. In our future work, we plan to focus on connectors to provide additional metrics. Our intuition is that metrics on connectors would emphasize aspects related to redundancy of communication paths, and therefore availability, resilience and denial of service prevention.

REFERENCES

Alshammari, B., Fidge, C., and Corney, D. (2009). Security Metrics for Object-Oriented Class Designs. In *Ninth International Conference on Quality Software*.

Casola, V., De Benedictis, A., Rak, M., and Villano, U. (2020). A novel Security-by-Design methodology: Modeling and assessing security by SLAs with a quantitative approach. *Journal of Systems and Software*, 163.

Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., and Jiang, Y. (2019). Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press.

Fernandez-Buglioni, E. (2013). *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley.

Gennari, J. and Garlan, D. (2012). Measuring Attack Surface in Software Architecture. Technical Report CMU-ISR-11-121, Carnegie Mellon University.

Herley, C. and van Oorschot, P. C. (2018). Science of Security: Combining theory and measurement to reflect the observable. *IEEE Security & Privacy*, 16(1).

Jürjens, J. (2002). UMLsec: Extending UML for Secure Systems Development. In *UML — The Unified Modeling Language*, LNCS. Springer.

Manadhata, P. K. and Wing, J. M. (2011). An Attack Surface Metric. *IEEE Transactions on Software Engineering*, 37(3).

NIST (2020). Security and Privacy Controls for Information Systems and Organizations.

Rak, M. (2017). Security Assurance of (Multi-)Cloud Application with Security SLA Composition. In *Green, Pervasive, and Cloud Computing*, LNCS. Springer.

Siavvas, M., Kehagias, D., Tzovaras, D., and Gelenbe, E. (2021). A hierarchical model for quantifying software security based on static analysis alerts and software metrics. *Software Quality Journal*, 29(2).

Skandylas, C., Khakpour, N., and Cámara, J. (2022). Security Countermeasure Selection for Component-Based Software-Intensive Systems. In *IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*.

Sommerville, I. (2016). *Software Engineering*. Pearson, 10th edition edition.

Wagner, S., Goeb, A., Heinemann, L., Kläs, M., Lampasona, C., Lochmann, K., Mayr, A., Plösch, R., Seidl, A., Streit, J., and Trendowicz, A. (2015). Operationalised product quality models and assessment: The Quamoco approach. *Information and Software Technology*, 62.

Waidner, M., Backes, M., and Müller-Quade, J. (2014). Development of Secure Software with Security By Design. Technical Report SIT-TR-2014-03, Fraunhofer Institute for Secure Information Technology.

Yskout, K., Scandariato, R., and Joosen, W. (2015). Do security patterns really help designers? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press.