Empirical Software Engineering (2025) 30:117
https://doi.org/10.1007/s10664-025-10649-7

A taxonomy of functional security features and how they can be located

Kevin Hermann[1] · Simon Schneider[2] · Catherine Tony[2] · Asli Yardim[1] · Sven Peldszus[1] · Thorsten Berger[1,3] · Riccardo Scandariato[2] · M. Angela Sasse[1] · Alena Naiakshina[1]

Accepted: 25 March 2025
© The Author(s) 2025

Abstract
Security must be considered in almost every software system. Unfortunately, selecting and implementing security features remains a challenge due to the wide variety of security threats and possible countermeasures. While security standards are intended to help developers, they are usually too abstract and vague to help implementing security features, or they merely help configuring such. A resource that describes security features at an abstraction level that lies between high-level (i.e., rather too general) and low-level (i.e., rather too specific) security standards could facilitate secure systems development. This resource should support the selection of appropriate security features to achieve high-level security goals, allow easy retrieval of relevant low-level details, and provide pointers to suitable ways to realize the security features. To realize security features, developers typically use external security libraries or frameworks, to minimize implementation mistakes. Even when using libraries, developers still make mistakes when writing code to integrate them, often resulting in security vulnerabilities. When security incidents occur or the system needs to be audited or maintained, it is essential to know what security features have been implemented and, more importantly, where they are located. This task, commonly referred to as feature location, is often tedious and error-prone. While dedicated feature location techniques exist, they require significant manual effort or adherence to strict development processes, preventing their use. Therefore, we have to support long-term tracking of implemented security features. We present a study of security features presented in the literature and their coverage in popular security frameworks. We contribute (1) a taxonomy of 68 functional implementation-level security features including a mapping to widely used security standards, (2) an examination of 21 popular security frameworks concerning which of these security features they provide, and (3) a discussion on the representation of security features in source code. Our taxonomy aims to aid developers in selecting appropriate security features and security frameworks, as well as relating them to security standards when they need to choose and implement security features for a software system.

Communicated by: Feldt and Zimmermann

Kevin Hermann, Simon Schneider, Catherine Tony and Asli Yardim These authors contributed equally to the paper.

Extended author information available on the last page of the article

Published online: 28 May 2025

Keywords Security · Security features · Security frameworks · Feature location · Security standard

1 Introduction

Considering security when developing software is crucial. Software vulnerabilities pose a major threat to the operation of software systems (Bau et al. 2012; Egele et al. 2013; Lazar et al. 2014; Nadi et al. 2016; Fahl et al. 2013; Krombholz et al. 2017; Roth et al. 2021). For example, in 2020, an entire hospital had to be shut down due to a successful attack on its IT systems, preventing access to patient data (BBC 2020). Unfortunately, considering the wide variety of threats and implementing appropriate countermeasures to create a secure design for a software system requires special expertise (Oyetoyan et al. 2016, 2019).

Security standards were created to help selecting appropriate security measures to protect software systems from threats. Unfortunately, their support for realizing *security features*—the concrete implementations of security measures in code—is limited.

Such standards are often too abstract and rather focus on the development process, on non-functional security requirements (e.g., the criticality of data), or on low-level details, such as specific implementation aspects of cryptography. While security design patterns exist to help implementing non-functional security features (e.g., secure logging pattern), developers lack guidance in selecting and implementing *functional security features* (e.g., authentication or encryption) to achieve security goals. Specifically, a functional security feature is a functionality of a software systems that aims to mitigate an attack, the impact of one, or to protect an asset.

Engineering functional security features is challenging. **First, developers lack an overview of functional security features**. Such an overview should facilitate selecting security features from both the high-level security goals considered by many security standards and from the many low-level details of how to implement specific features securely. **Second, after selecting suitable functional security features, developers need to implement them**, typically by incorporating them from a security library or framework (Hermann et al. 2025). Unfortunately, a systematic overview of what security features are offered by which library or framework is missing. Developers, therefore, often fall back on the ones they already know. However, depending on the project, choosing a different security framework would allow using libraries that might provide better-suited implementations of security features. Even when using security libraries, security issues often arise in the manually implemented parts of applications, e.g., due to the insecure use of libraries (Acar et al. 2017) or bad usability (Patnaik et al. 2019). Fixing new vulnerabilities requires developers to review and fix them quickly once they are discovered (Russo et al. 2019; Peldszus et al. 2021). **Third, it is important to know what security features are implemented in a system at hand, and where they are located.** Many security standards, such as the *Common Criteria* (CC) (ISO/IEC JTC 1/SC 27 2009) or the *ISO/SAE 21434* for road vehicles (ISO/TC 22/SC 32 2021), require maintaining and tracing security features. Unfortunately, today's traceability techniques require significant manual effort, even when using tools, such as DOORS (IBM 2023b). Others depend on strict development processes and impose high overhead (Peldszus 2022). When features are not recorded or maintained properly, recovering them is laborious and error-prone (Biggerstaff et al. 1994; Dit et al. 2013; Krueger et al. 2019; Rubin and Chechik 2013). Recording features during development, when the feature is still fresh in the

mind of the developer (Seiler and Paech 2017; Ji et al. 2015; Martinson et al. 2021; Bergel et al. 2021; Schwarz et al. 2020; Entekhabi et al. 2019; Andam et al. 2017; Mukelabai et al. 2023), is rarely done in practice. While automated feature location techniques exist, they are difficult to use and produce too many false positives (Rubin and Chechik 2013; ben Othmane et al. 2015; Cornell 2012; Hewett and Kijsanayothin 2009; ben Othmane et al. 2017; Abukwaik et al. 2018) to be relevant in practice. Improving our empirical understanding of how security features are represented in security frameworks, using what mechanisms (e.g., configuration options, code annotations, or APIs), would help to build better methods and tools to locate security features in code.

In summary, supporting the development of secure software systems requires effective methods and tools for selecting, implementing, and locating security features in code bases. The different granularities at which security features can be considered is still an open challenge (Peldszus 2022), as well as their scattering over the code base and cross-cutting nature. While high-level security features are often hard to locate, as they are implemented across the codebase, locating fine-grained security features requires intricate knowledge that many developers lack. It is unclear yet, at which level of granularity security features manifest in implementations, preventing the development of lightweight support. Even security standards do not provide an adequate level of abstraction to be effectively used by developers for selecting which security features must be implemented to reach desired security goals. What is missing is a systematic representation of what functional security features exist, accompanied by a description suitable for developers, and a mapping to relevant security standards. We aim to improve the understanding of implementation-level security features and explore the following research questions:

RQ1: What functional implementation-level security features are considered in the literature?

RQ2: What functional implementation-level security features are provided by security frameworks in practice?

RQ3: Which functional implementation-level security features can be located by leveraging information from security frameworks?

We addressed these research questions as follows. First, we established a taxonomy of functional implementation-level security features by reviewing literature that systematically describes security features. Second, we mapped the taxonomy to four generally recognized and well-established security standards: the *ISO/IEC 27000* series, the *Common Criteria (CC)*, the *NIST SP800-53*, and the *NIST Cybersecurity Framework*. Third, we investigated state-of-the-art security frameworks as discussed by developers on platforms such as Stack Overflow and Reddit. Finally, we explored the mechanisms used by the frameworks for providing security features to developers and how these can be used for locating security features within applications. We demonstrate that our taxonomy can be related to all functional security features from popular security standards. Further, we demonstrate that security features from security frameworks target all security aspects covered in our taxonomy but do not capture the more detailed concepts considered in the literature. Finally, we show that security frameworks offer an entry point for locating security features through their API, configuration file and annotations, but still require considering additional code, that is required to integrate them.

2 Background and related work

We now discuss the notion of security features to motivate our work and introduce the necessary background.

2.1 Running example

As a motivating example for this work, we consider a simplified electronic health record system (EHRS) for a hospital.

In hospitals, many different groups of people are involved in treating a patient. Treatment requires data from a variety of sources, such as a diagnosis from a physician, health measurements collected by nurses, or data from specialists such as radiologists. An EHRS enables the capture and analysis of medical data, but often includes additional supporting functionality such as appointment management, medical data analysis, and administrative support such as billing. As illustrated in Fig. 1, our EHRS assumes several groups of users, which interact differently with the system: Patients, Doctors, Nurses, Administrative Hospital Staff, and many more. A doctor can store a diagnosis or an examination report for a patient within the EHRS. Similarly, nurses store measurements such as data about body temperature or blood pressure within the system. While doctors and nurses can store data for a patient, only a designated doctor, chosen by the patient, can retrieve the data to e.g., plan further treatment. Administrative hospital staff can store and retrieve data related to billing within and from the system.

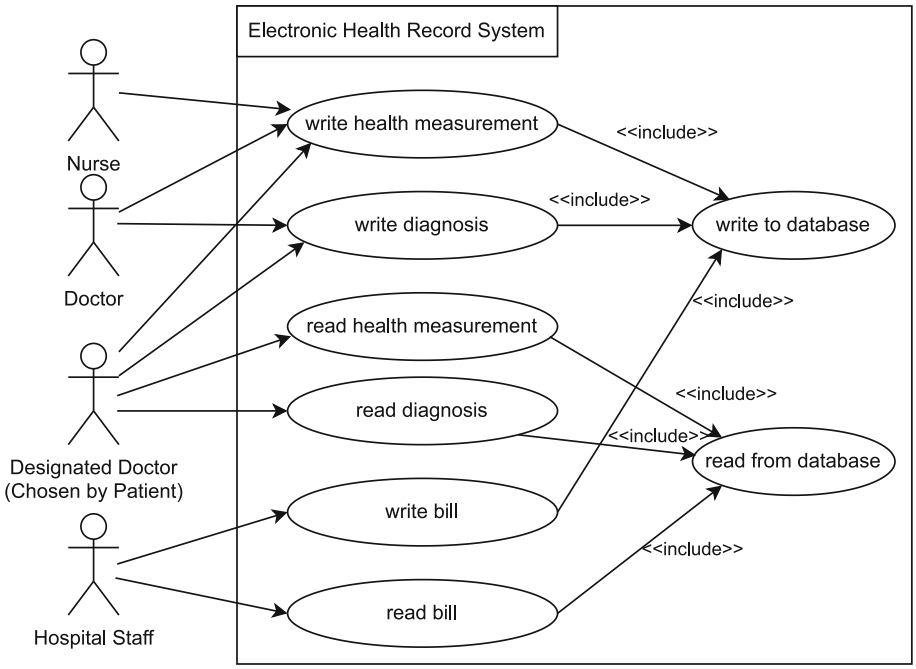

Fig. 1 Use case diagram illustrating how different users store and retrieve data from our simplified exemplary EHRS

Since an EHRS manages sensitive data such as patient data, hospital data, or diagnosis data that is used to decide about the treatment of patients, the system must preserve data integrity and confidentiality at all times. Also, availability must be ensured to allow the hospital to operate—in 2020 a hospital was shut down due to an attack, and new emergency patients had to be relocated to other hospitals further away (BBC 2020).

In summary, our simplified exemplary EHRS allows doctors to plan the treatment of patients by storing all related data in a central data storage. Due to the sensitivity of this data, it must be securely stored, and access must be controlled to ensure that only authorized personnel is allowed to view sensitive data of patients. This requires considering a wide range of security features to be implemented into the system.

2.2 Security features

A feature is a distinct label representing the capabilities or behaviors of software systems (Berger et al. 2015). A feature can be seen as *"a logical unit of behavior specified by a set of functional and non-functional requirements"* (Bosch 2000). For example, every use case in Fig. 1 can be seen as a required feature of the EHRS. A feature can also be defined as a characteristic, that distinguishes a system from other systems within a family of related systems (Batory et al. 2004). Other definitions describe a feature as a user-visible aspect of a system (Chen et al. 2005; Kang et al. 1990) or an aspect that increases value for a customer (Riebisch 2003).

In this work, we use the notion of *security features*, which provide functionalities that address security issues by preventing a security attack or realizing a security requirement (McGraw 2004). A typical example of a security feature is the authentication of the different users of the outlined EHRS. Security features must be carefully planned, even at the architecture level, since missing security features can lead to severe weaknesses in software systems (Santos et al. 2017, 2019). While security features could realize, e.g., non-functional requirements (Potter and McGraw 2004), we focus on functional security features, which are security measures that manifest in the code base and address a functional requirement of a software system. On the other hand, an example of a non-functional security feature that is not in the scope of our work is the secure design pattern of distrustful decomposition (Dougherty et al. 2009).

2.3 Security feature taxonomies and ontologies

There are several works that organize software security-related concepts into constructs including taxonomies and ontologies to show how they are interrelated. For instance, a work by Tsipenyuk et al. 2005 presents a taxonomy of coding errors and configuration issues that lead to security vulnerabilities. The main aspects covered in this taxonomy include *input validation and representation, API abuse, security features, time and state, errors, code quality, encapsulation, and environment*. Even though this taxonomy has a dedicated section for security features, it only covers 9 features, 5 of which are related to password management. A taxonomy for cloud systems security (Habiba et al. 2014) organized the security features into categories such as *authentication, authorization, identity federation, privacy, user-centricity, logging, and editing* that are essential for cloud-based identity management systems. Security aspects of the Internet of Things (IoT) domain are also discussed in the literature such as by Khanam et al. (2020), who presented a taxonomy of IoT security attacks in physical, network, and application layers along with their corresponding countermeasures. Another

work (Blythe et al. 2019) analyzed the user manuals and support pages of IoT devices to collect security features such as *two-factor authentication, product lock, and local communication encryption* provided by consumer IoT products. Similarly, there are also several other works that organizes security aspects related to cloud security (Hendre and Joshi 2015; Bhatia and Verma 2017), web services (Denker et al. 2003; Kim et al. 2007; Busch and Wirsing 2015), information security (Venter and Eloff 2003; Herzog et al. 2007; Vorobiev and Bekmamedova 2010), and IoT (Abbas et al. 2005; Herzog et al. 2007) into taxonomies and ontologies.

Although these studies provide valuable insights into security across various fields, they are often domain-specific and largely focus on attacks and vulnerabilities, offering limited comprehensive lists of security features for developers to reference during software development. Therefore, extracting and consolidating the security features discussed in these works into a single, accessible resource would benefit developers by providing a centralized reference during software development.

2.4 Feature location

To maintain and evolve features, developers need to know their location in the code base at hand (Ji et al. 2015). Feature location is the process of identifying the code that implements a particular feature (Revelle et al. 2005). As such, it is one of the most common activities of developers. Unfortunately, feature location is laborious and error-prone, especially for long-living software systems with many developers and features that are scattered over the code base. Documenting features would help, but requires upfront effort and is often avoided, requiring recovery of features and their locations (Rubin and Chechik 2013).

Feature location classifies into eager and lazy strategies (Ji et al. 2015). The *eager strategy* refers to recording information on feature locations during their development, either directly within the software assets or in external trace databases. Different methods exist, such as using embedded code annotations for recording features, together with tools for browsing/visualizing features (Seiler and Paech 2017; Martinson et al. 2021; Andam et al. 2017; Bergel et al. 2021; Entekhabi et al. 2019), as well as feature traceability databases, such as FEAT (Robillard and Murphy 2007). In contrast, the *lazy strategy* recovers feature locations when needed. Both, manual (Krueger et al. 2019) and automated (Rubin and Chechik 2013) techniques have been explored in research. However, manual recovery is laborious and error-prone, and automated techniques (often relying on natural-language processing or machine-learning methods) yield too many false positives to be usable in practice. As such, our long-term goal is to establish methods and tools to record security features eagerly. However, to construct effective techniques, we need to improve our empirical understanding of what security features are and how they manifest in source code. In other words, developers need to know what security features are traceworthy and on which level of abstraction they should be captured—the goal of our study. In addition, shedding light on what security features can be located easily in the implementation can also help improve manual and automated feature-location methods that try to retroactively recover features from software assets.

2.5 Security feature tracing

The interrelation of features and their implementation in code throughout the development process is called tracing. It is often required by security standards such as the Common Criteria

(ISO/IEC JTC 1/SC 27 2009). To this end, previous work proposes techniques to enable the traceability of security features. The technique SecSTAR by Fang et al. (2012) traces a software system's security structure and properties and generates diagrams to support security analysis. Enterprise Architect (Sparxsystems 2023) provides commercial tool support for strictly coupling UML models to code to facilitate the synchronization between them, which could also be used for UML models describing security features. SecReq (Houmb et al. 2010) is a methodology for eliciting security requirements as well as the early detection and refinement of security issues with traceability support for UML design models. Islam et al. (2011) propose a framework for obtaining security requirements from laws and regulations and tracing them to security requirements throughout the whole development life cycle to enable checking compliance with laws and regulations. The GRaViTY (Peldszus 2020, 2022) framework maintains traceability between different artifacts, such as UML models, Java source code, and program models. It uses trace links to propagate security requirements into the implementation. Strong coupling between the source code and the models is required to enable the traceability of security features using these approaches. In summary, these approaches do not yet provide enough flexibility for a vast practical application.

2.6 Security standards and guidelines

Security standards and guidelines provide developers with security features that need to be realized to secure a software system. Many product requirements in the industry are formulated around security standards, for example, a system should adhere to all certification requirements of a specific standard. In fact, standards compliance is mandatory for systems like the EHRS (European Parliament and Council of the European Union 2007; United States Congress 1996). An organization's information security management system or a single software system can be certified according to a certain security standard if it can be proven that the required security controls are implemented. Such proofs are usually in the form of documentation of carried-out activities, e.g., the identification of security threats and the specification and realization of mitigating security features. Due to the procedural nature of the standards, the requirements, for the most part, describe actions that have to be performed or high-level security functionality that has to be achieved. The few implementation-level security features that are mentioned are mostly in terms of specific technologies that are given as examples of how to realize some security control and are often lost in a huge body of text.

For illustration, Figure 2 shows an excerpt from the NIST SP 800-53 standard which provides security and privacy controls for information systems and organizations. The excerpt focuses on security controls for *contingency planning*, such as *system backup* and presents associated control enhancements that add functionality or specificity to this base control. It can be seen that the functional-level security features such as *cryptographic protection* are hidden among several other security-related information such as *testing for reliability and integrity, test restoration using sampling* and so on. The figure also shows multiple cross-references (e.g., SC-12, SC-13, SC-28) meant to provide additional details on the control obscuring specific functional-level security features in an extensive and interconnected array of information. Additionally, the descriptions for such security features such as "*implement cryptographic mechanisms to prevent unauthorized disclosure,*" as in the figure, are often broad and abstract providing little concrete guidance for its practical implementation. Therefore, we see the need for a comprehensive overview of implementation-level functional security features. A taxonomy of such features, together with a mapping to the standards and

3.6 Contingency Planning — **Controls**

CP-9 SYSTEM BACKUP — **Control ID**

Control:
a. Conduct backups of user-level information...
...
d. Protect the confidentiality, integrity, and availability of backup information. — **Goal**

Discussion: **[108 Words]**
Control Enhancements: — **Description**

(1) SYSTEM BACKUP | TESTING FOR RELIABILITY AND INTEGRITY
Test backup information [Assignment: organization-defined frequency] to verify media reliability and information integrity. — **Enhancement: Testing**

...

(2) SYSTEM BACKUP | TEST RESTORATION USING SAMPLING
Use a sample of backup information in the restoration of selected system functions as part of contingency plan testing. — **Enhancement: Testing**

...

...

(8) SYSTEM BACKUP | CRYPTOGRAPHIC PROTECTION
Implement cryptographic mechanisms to prevent unauthorized disclosure and modification of [Assignment: organization-defined backup information].
Discussion: The selection of cryptographic mechanisms is based on the need to protect the confidentiality and integrity of backup information. The strength of mechanisms selected is commensurate with the security category or classification of the information. ... — **Enhancement: Security Feature**

Related Controls: SC-12, SC-13, SC-28.

3.18 System and Communications Protection — **Controls**

SC-13 CRYPTOGRAPHIC PROTECTION — **Control ID**

Control: **[31 words]** — **Goal**

Discussion: **[118 words]** — **Description**

Related Controls: AC-2, AC-3, AC-7, AC-17, AC-18, AC-19, AU-9, AU-10, CM-11, CP-9, IA-3, IA-5, IA-7, MA-4, MP-2, MP-4, MP-5, SA-4, SA-8, SA-9, SC-8, SC-12, SC-20, SC-23, SC-28, SC-40, SI-3, SI-7. — **Cross References**

Fig. 2 Excerpt of the NIST SP 800-53 standard for security and privacy controls for information systems and organizations

guidelines, could assist developers by giving actionable advice for how to realize required security controls.

3 Methodology

We conducted a *systematic review* (Ralph et al. 2021) of literature and security frameworks to elicit functional security features and how they are provided in security frameworks. Figure 3 shows our research methodology. To identify implementation-level security features, we reviewed the literature that presents structured collections of security features (**RQ1**). To ensure the applicability of the taxonomy in practice and to validate it, we created a mapping between our taxonomy and security features described in widely used security standards and potentially adapted the taxonomy. Additionally, we collected and inspected existing security frameworks discussed by developers to understand which functional implementation-level

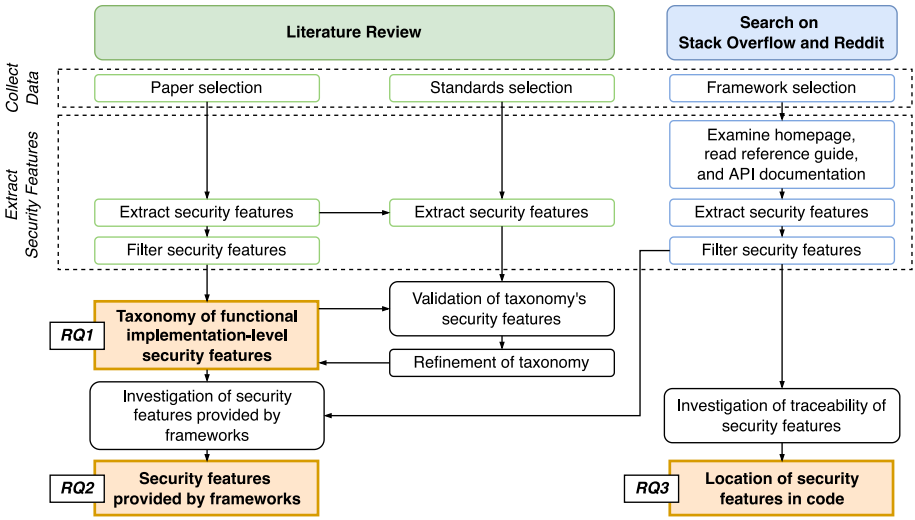

Fig. 3 Overview of the applied research methodology

security features are provided to developers (**RQ2**) and investigated their representation in source code through different mechanisms (e.g., code annotations) (**RQ3**).

All steps in the creation of the taxonomy, the mapping to the security standards, and the analysis of the security frameworks followed the same general process, considering the recommendations by McDonald et al. (2019). In each case, two authors first conducted an exploratory analysis of the relevant artifacts (literature, standards, and frameworks) to establish a basis for an open discussion process. Then, in group meetings with the first five authors, the relevant elements (e.g., security features in the literature or standards) were identified based on the different views of the two authors who initially analyzed them, and their definitions were derived. To streamline this process, the two authors who analyzed the raw artifacts prepared proposals for elements and their definitions in cases where they deemed their observations to be closely related or complementary. However, all extracted content, including these proposals, underwent the same discussion process in the larger group. In more than 30 meetings lasting about one hour each, we further regularly discussed the resulting taxonomy until we reached full agreement, i.e., each decision was discussed until all involved authors agreed on the solution. Since we reached full agreement after our discussion rounds and reaching the agreement is the main purpose of the process but not its sole outcome, as the iterative discussions also served to refine perspectives and surface meaningful themes, we did not calculate an inter-coder agreement (McDonald et al. 2019).

3.1 Systematic literature review

To establish an empirical understanding of functional implementation-level security features, we reviewed structured collections of such in the literature.

3.1.1 Paper selection

We conducted a manual two-step screening process to select relevant papers, as shown in Fig. 4. We searched for relevant publications on Google Scholar using the tool *Publish or*

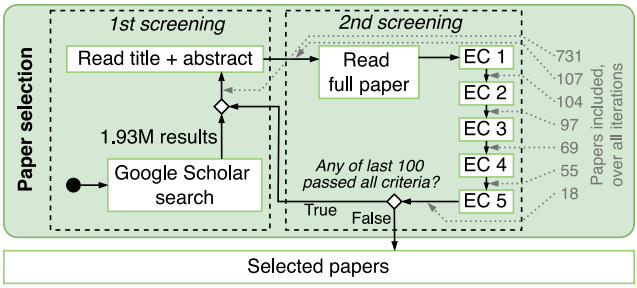

Fig. 4 Paper selection process of the SLR with sequential application of exclusion criteria

Perish (Harzing 2007). Google Scholar covers the typical major data sources for literature reviews such as IEEExplore or the ACM Digital Library. In a study by Valente et al. (2022), the authors observed that Google Scholar provides the most comprehensive search results for literature reviews in the computer science domain. By using a single data source, we could directly apply a saturation criterion without having to merge search rankings of different sources. To this end, five authors collected keywords to tailor the search in a group meeting: (1) terms related to the considered implementation level, (2) synonyms of "security," and (3) terms describing a systematic representation of aspects. Thus, we ended up with the following compound search term:

```
(implementation OR code OR program) AND
(security OR secure) AND (ontology OR taxonomy OR "body of
knowledge" OR
"system of knowledge" OR "conceptual model")
```

We performed a query with this search term and examined each result in their ranked order in a two-step screening process. In the first step, we read the title and abstract of the paper to verify that it includes a structured collection of security features. Whenever the title and abstract were not enough to make this decision, we read other parts of the paper. We considered 107 papers for further review in the next step. When a paper passed the first step, we read the full paper in the second step. We filtered the papers according to five criteria, that we chose to fit our scope of functional security features and applied all of them one after another to each resulting paper. Furthermore, we excluded papers that are only applicable in specific domains, as our goal is to provide an overview of security features in general software systems.

Through the investigation of the literature, we mostly encountered ontologies and taxonomies for structuring such collections. While a taxonomy represents a "general categorization based on a class/subclass relationships," an ontology is "the formal specification of domain concepts and their relationships" (Hakeem and Shah 2004). Additionally, a number of security standards describe security requirements, which indicate security features needed to fulfill the requirements.

Exclusion criteria:

EC1 : the paper is not published in a conference, or journal
EC2 : the collection of security features is not made available
EC3 : the scope of the paper is limited to a specific application domain, e.g., only CAN bus security

EC4 : only threats, vulnerabilities, risks, and so on are considered without presenting countermeasures
EC5 : the paper is not associated with functional security features considered in software engineering

For data analysis, we followed a process considering the recommendations by McDonald et al. (2019). Two authors initially performed the selection and repeatedly compared their results to check whether the saturation criterion was fulfilled. Then, the results including all possible conflicts and ambiguities were discussed in regular group meetings with the first five authors. We reached saturation at the mark of 731 search results, as we observed no new papers that passed the screening process within over 100 search results before that point. Table 1 lists the 18 papers that passed all exclusion criteria and were considered for extracting security features. As shown in Fig. 4, we excluded most papers (37) in the second screening step based on EC5. Note that we report only the first applicable exclusion criterion per paper, as we did not check the additional criteria after exclusion.

3.1.2 Extraction of security features

After shortlisting the papers through the two-step screening process, we collected all security features they present to facilitate a subsequent group discussion about the features. The search term and the exclusion criteria used in our SLR targeted papers presenting structured collections of security features. (see Section 3.1.1). Consequently, all selected papers contained a systematized presentation of security features. The process of extracting them thus consisted of identifying this collection of security features in each paper. Security features were represented either as a graph or a table in the papers' *Results* section (or comparable but differently named sections). Their identification was a straightforward process, but was performed by two researchers independently nevertheless to mitigate possible human errors. No deviations between the two researchers' identification of the security feature collections in the papers were observed. The extracted features were then discussed by the first five authors in recurring meetings. In the discussions, we removed any features that were not functional implementation-level security features related to software engineering. Additionally, we excluded security features that are only limited to specific application domains of software systems such as automotive systems to keep the resulting set of security features as widely applicable as possible. In particular, we removed all terms meeting any of the following properties:

- Specific to hardware (e.g., ID card or credit card)
- Limited to a single platform, such as operating systems, libraries, or other technologies (e.g., μC/OS)
- Not related to security (e.g., supplier or memory)
- Associated with security attacks or vulnerabilities (e.g., DoS attack, sniffing attack or P2P attack)
- Restricted to a single application domain (e.g., the automotive domain)

We created a taxonomy containing all the collected security features. With a graph editor, we compared the presented security concepts and identified overlaps, i.e., security features contained in multiple of the analyzed papers. We merged the individual sets of terms of each of the selected papers, starting from one paper and iteratively adding the others by identifying security features included in the merged set and the newly added paper and adding all connected features at this place. Finally, we classified and grouped the security features

Table 1 Shortlisted papers presenting security features

ID	Authors	Title	Year
P1	Venter et al.	A taxonomy for information security technologies (Venter and Eloff 2003)	2003
P2	Denker et al.	Security for DAML web services: annotation and matchmaking (Denker et al. 2003)	2003
P3	Abbas et al.	A state of the art security taxonomy of internet security: threats and countermeasures (Abbas et al. 2005)	2005
P4	Herzog et al.	An Ontology of Information Security (Herzog et al. 2007)	2007
P5	Kim et al.	Security Ontology to Facilitate Web Service Description and Discovery (Kim et al. 2007)	2007
P6	Vorobiev et al.	An Ontology-Driven Approach Applied to Information Security (Vorobiev and Bekmamedova 2010P	2010
P7	Kang et al.	A Security Ontology with MDA for Software Development (Kang and Liang 2013)	2013
P8	Habiba et al.	Cloud identity management security issues & solutions: a taxonomy (Habiba et al. 2014)	2014
P9	Hendre et al.	A semantic approach to cloud security and compliance (Hendre and Joshi 2015)	2015
P10	Busch et al.	An ontology for secure web application (Busch and Wirsing 2015)	2015
P11	Talooki et al.	Security concerns and countermeasures in network coding based communication systems: A survey (Talooki et al. 2015)	2015
P12	Kaur et al.	Security of software-defined networks: Taxonomic modeling, key components and open research area (Kaur et al. 2016)	2016
P13	Bhatia et al.	Data security in mobile cloud computing paradigm: a survey, taxonomy, and open research issues (Bhatia and Verma 2017)	2017
P14	Adat et al.	Security in Internet of Things: issues, challenges, taxonomy, and architecture (Adat and Gupta 2018)	2018
P15	Harbi et al.	A review of security in internet of things (Harbi et al. 2019)	2019
P16	Kumar et al.	On cloud security requirements, threats, vulnerabilities, and countermeasures: A survey (Kumar and Goyal 2019)	2019
P17	Khanam et al.	A survey of security challenges, attacks taxonomy and advanced countermeasures in the internet of things (Khanam et al. 2020)	2020
P18	Mahapatra et al.	A Survey on Secure Transmission in Internet of Things: Taxonomy, Recent Techniques, Research Requirements, and Challenges (Mahapatra et al. 2020)	2020

to give them a coherent structure, following the classification and hierarchy rules from the originating papers. Five authors discussed the taxonomy's terms to agree on a structure. The resulting taxonomy contains all implementation-level security features identified in the final set of papers of our SLR.

3.1.3 Mapping to security standards

Security standards are generally regarded as highly reliable sources of information for securing software systems because they undergo rigorous review processes before publication. Despite the lack of implementation details, official standards and guidelines issued by large, reputable organizations are a common source of information about software security. Thus, creating a mapping of our taxonomy to established security standards increases its relevance for application in the industry. Furthermore, a successful mapping allows reasoning about the validity of the derived taxonomy.

We expect that each functional security feature in the standards can be mapped to one or more security features in the taxonomy. Therefore, the mapping allowed us to validate the completeness of the taxonomy we derived from the literature. As relevant standards for the mapping, we analyzed the *ISO/IEC 27000 family*, the *Common Criteria* (CC), the *NIST SP800-53*, and the *NIST Cybersecurity Framework*, which are widely recognized in the industry as the most important security standards and guidelines.

To create the mappings, two authors independently analyzed each section of the standards, identifying functional security features and matching them to the corresponding proposed security features from the taxonomy based on the description provided in the standards. In addition, for identified security features that were not yet part of the taxonomy, they also proposed adaptations to the taxonomy to support all functional security features from the standards. Then, together with three further authors, each part was discussed, and a decision for the mapping was taken collaboratively. To this end, for the security features that could not be immediately mapped to the taxonomy, the first five authors of this work discussed whether, where, and how to adapt the taxonomy to include the features. This validation and adaptation process was performed for each standard, starting with the CC.

We report in Section 4.2 for each standard how well it could be mapped to the taxonomy, and what minor and major changes to the existing taxonomy were required to allow the mapping of all security features. In this way, we provide guidance to developers who can use the taxonomy as an abstraction of the standards. The granularity of security features in our taxonomy lies between the high-level descriptions of security mechanisms found in most standards and the detailed requirements for specific technologies found in others.

3.2 Identification of security frameworks

We systematically identified popular security frameworks discussed on the popular developer platform Stack Overflow and the programming community of Reddit to compare the state-of-practices of functional security features with our derived taxonomy. We chose Stack Overflow since it is one of the largest and also most popular platforms for content related to software development amongst developers (Xia et al. 2017). On Stack Overflow, developers mainly discuss problems or seek recommendations when facing problems during their development tasks. As a second data source, we chose Reddit's largest developer community "r/programming", which, from its origins, is the most popular place on the platform for exchanging programming related content. In contrast to Stack Overflow, developers do not discuss the usage of security frameworks, but present them to other developers by sharing articles or repositories, allowing us to capture a different type of discussion. We investigated which implementation-level security features are provided by frameworks used in practice

and their relation to the literature captured in our taxonomy. Further, we investigated the mechanisms used to provide security features and how these could be leveraged for locating security features. We refer to *security frameworks* when they focus on providing security mechanisms and related functionality.

3.2.1 Identifying security frameworks from stack overflow and reddit

To identify relevant security frameworks discussed in practice, we searched for *"security framework"* on the widely used developer discussion platform *stackoverflow.com*. We used the Stack Exchange API v2.3 (Stack Exchange 2022) to download threads, ensuring that they remained unaltered throughout the entire analysis period when we reviewed the results. Two authors sorted the threads by relevance and investigated the results by independently reading the questions, answers, and comments of each thread. In the threads, we manually searched for mentions of security frameworks or security modules of general frameworks. We continued the search until no new frameworks were mentioned in the last 20 threads. We reached this data saturation (Glaser 1978) at 250 threads.

For the search on Reddit, we employed a similar approach as we did for the search on StackOverflow. Searching for *"security framework"* resulted in 249 threads. On Reddit, threads contain comments and either a user created discussion, or a link to an article. As an initial filtering step, two authors exhaustively and independently read the titles of each thread, including all threads discussing security or securing applications for further investigation. They then merged their sets of included threads, resulting in 68 threads. Afterwards, they read all threads, including linked articles and comments, to extract all mentioned security frameworks. Finally, we merged the results with our Stack Overflow search.

3.2.2 Extracting security features from security frameworks

To derive a final list of relevant frameworks, we selected all security frameworks that were mentioned in at least two threads during the identifications of security frameworks in the merged results. We examined the selected frameworks in depth to capture the provided security features. To this end, two authors used three different sources of information for each framework (unless not provided for a specific framework), the related *homepage*, a *reference guide*, and the *official documentation*. Each author independently recorded security features described in each source. The framework's homepage usually provided a general overview of the security features included in the framework, while in the reference guide, a more detailed look at the security features was often given. Using the official documentation, we investigated the low-level components and encountered an in-depth description of the framework and its methods. In cases where the three sources used different terminology to describe the same security feature, the two authors compared the terminology and descriptions across the sources in joint sessions, and chose the term used by at least two, or the best-fitting one if all three used different terms. Furthermore, in these discussions we categorized some specific terms, such as *username* and *password*, into broader security features such as *credentials*. We considered any security feature that offers a reusable functionality at the implementation level that addresses a security requirement or security issue (McGraw 2004).

The same two authors organized the features into a hierarchy based on the structure in the frameworks' documentation. Discrepancies were discussed and resolved through collaborative sessions, ensuring that the resulting hierarchy accurately reflected the frameworks'

intended structure. While investigating the security features, we documented in parallel information on using the individual security features offered by the selected frameworks in source code. Based on the mechanism described in the documentation, we grouped the security features into the three realization methods *annotations*, *APIs*, and *configuration files*. Any disagreements were addressed by re-examining the documentation together to reach a consensus. Whenever a security feature was mentioned in combination with an API artifact, such as a method, interface, variable, or class, we grouped the security feature to a realization with an API. Likewise, if a configuration file, such as a `.xml`, `.properties`, or `.conf` file was mentioned along the security feature, we mapped the realization to a configuration file. Finally, we applied the same procedure for annotation mechanisms, such as Java annotations or attributes in C#. We collected this information for each security feature and framework to reason about their traceability (RQ3).

4 Taxonomy of implementation-level security features (*RQ1*)

In our SLR, we identified papers that present ontologies and taxonomies of software security features from which we extracted functional code-level security features. Thereafter, we constructed a taxonomy out of these and mapped it to four security standards to further validate and refine it. In the following, we describe the results of our analysis.

Table 1 presents the 18 papers (referred to as *P1 - P18* in this work) identified in our SLR for instantiating our taxonomy of functional security features. Our taxonomy consists of 68 implementation-level security features shown in Fig. 5. Note that security features are not necessarily mutually exclusive from each other. As such, they may be combined to achieve a higher security goal or property (e.g., a system might realize both credentials and multifactor authentication to protect the confidentiality of data). Security features with a similar goal or security property that they achieve were grouped under top-level security features, which we identified from the hierarchies of the reviewed ontologies and taxonomies. We additionally added annotations to the corresponding security features to indicate the frameworks and standards in which they were identified (see Fig. 6, and Fig. 8 to Fig. 11).

4.1 Taxonomy of functional security features

We identified five top-level security features, as shown in Fig. 5: *access control*, *cryptography*, *security monitoring*, *secure data handling* and *system state protection*. Table 2 shows the occurrences of these security features in the papers. Four papers included all top-level features, however, our taxonomy contained more security features beyond the top-level ones for each of the papers. The top-level feature *access control* is included in all 18 papers. Only one paper does not include *cryptography*. This shows the importance of these two groups of features. The detailed taxonomy of these security features is presented in Fig. 5. We now describe each of them in detail.

4.1.1 Access control

Access control covers security features that are concerned with regulating access to protected resources and granting them only to authorized subjects. For example, in the EHRS from our example, doctors should only be allowed to read sensitive data of a patient they are designated to, while other doctors are only allowed to write to it. However, doctors have the permission

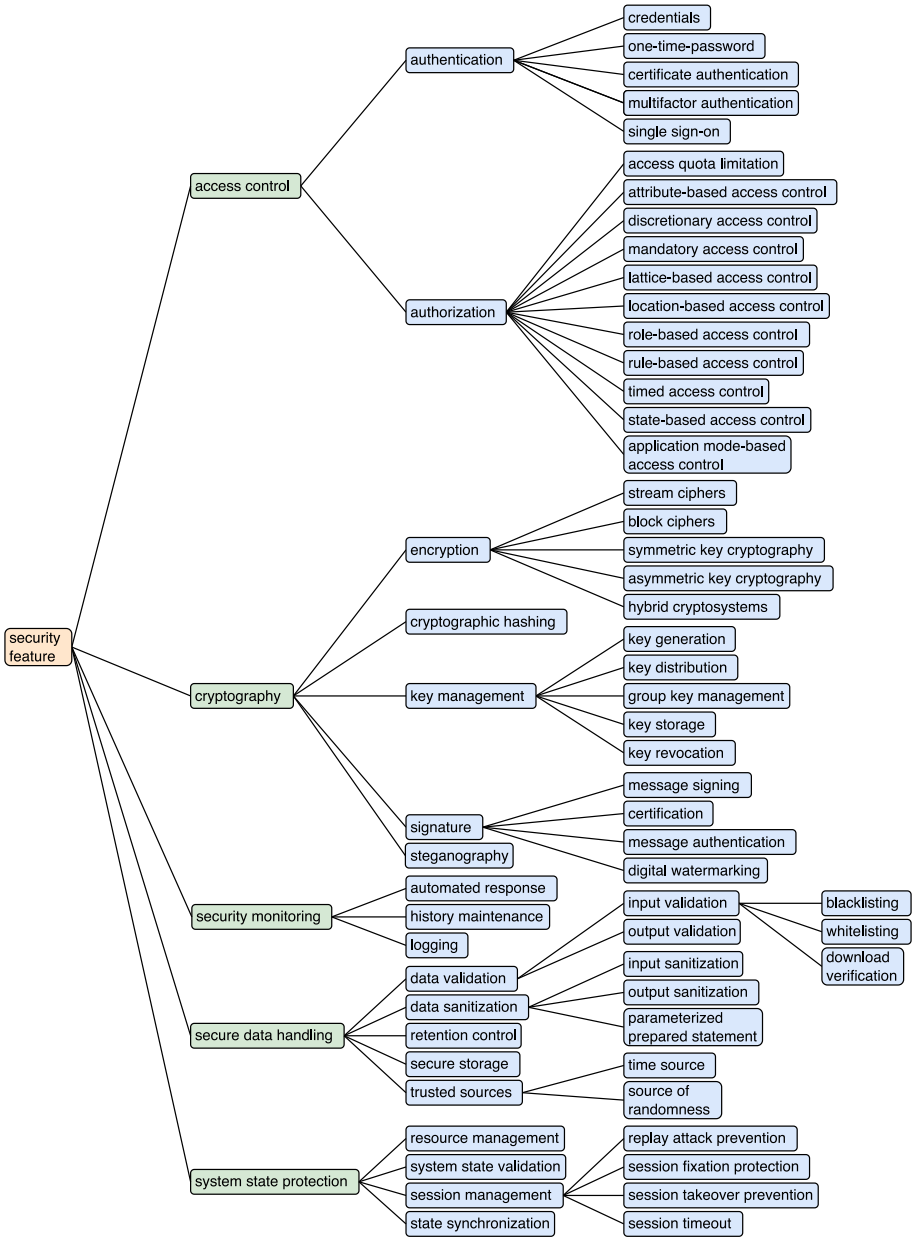

Fig. 5 Taxonomy of implementation-level security features

to view anonymized statistics, such as past treatment of patients for different diagnoses. This requires the control of all accesses to the system. All papers include security features for access control in their ontologies. The top-level feature comprises two major blocks of features, one grouped under the sub-feature *authentication* and one under *authorization* (Fig. 6).

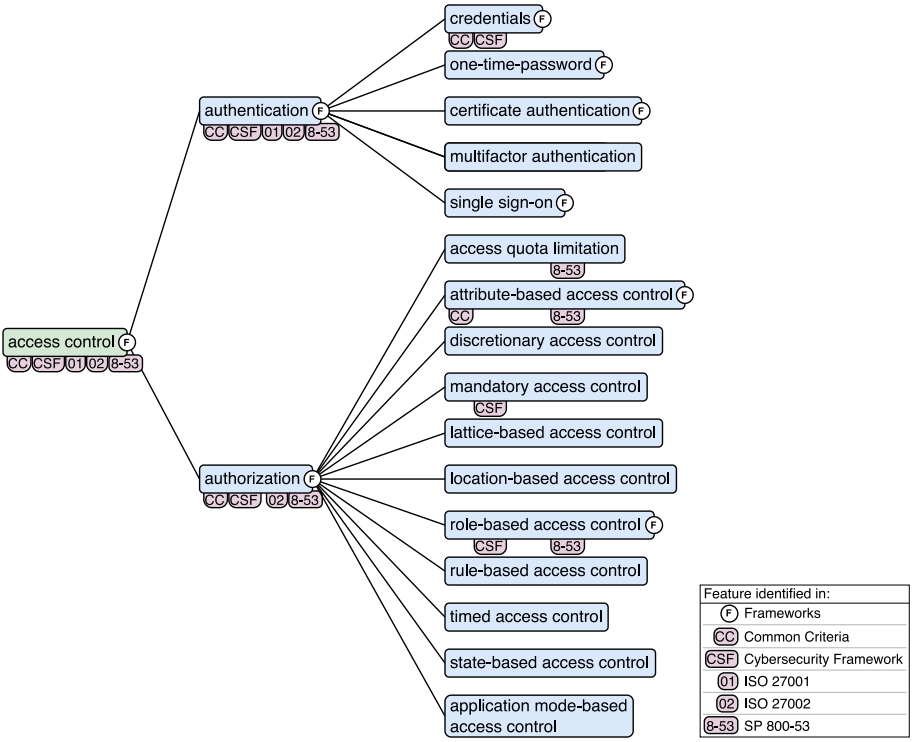

Fig. 6 Sub-features of the top-level security feature *access control*

Authentication is presented in the papers both, as a security notion, objective, or a means to achieve data confidentiality, and as a security feature that implements these. The user must be identified, i.e., whether it is a doctor or not, before they gain access to it. *P2*, *P5*, *P12*, *P13*, and *P14* describe authentication as the identification and verification of a party sending a request to a network or application, where the associated security features support the realization of this. Here, authentication is a security feature that often entails multiple other security features because of its complexity or manifests at multiple places in the code base. According to *P7*, authentication is used to achieve data confidentiality, while the remaining papers define authentication in terms of user, data, and message integrity/authenticity. In our taxonomy, authentication classifies into more specific security features (see Fig. 6). When authentication is performed using *Credentials*, data objects such as usernames or passwords are used to verify the identity of a user. In the EHRS from our example, each hospital staff member could receive a set of credentials from an administrator, allowing them to log into the system from an arbitrary device within the hospital. *One-time-password* is a method where authentication is performed with randomly generated temporary passwords (Habiba et al. 2014). *Certificate authentication* refers to authentication using certificates such as X.509 (Denker et al. 2003). *Multifactor authentication* requires the user to provide more than one way of authentication (Bhatia and Verma 2017). *Single sign-on* allows users to securely log in to multiple systems using only one set of credentials.

Table 2 Security feature occurrence in the 18 shortlisted papers

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18
access control																		
authentication	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
authorization			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
cryptography																		
cryptographic hashing			✓			✓				✓	✓		✓	✓	✓	✓	✓	✓
encryption	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
key management		✓	✓	✓	✓	✓			✓	✓	✓		✓	✓	✓	✓	✓	✓
signature	✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓
steganography				✓						✓								
security monitoring																		
automated response									✓					✓		✓		
history maintenance								✓								✓		
logging	✓			✓				✓	✓	✓	✓		✓	✓		✓	✓	
secure data handling																		
data validation										✓	✓		✓			✓		
data sanitization										✓						✓		
retention control										✓								
secure storage										✓	✓		✓	✓	✓	✓	✓	✓
trusted source										✓			✓		✓	✓	✓	
system state protection																		
resource management											✓			✓		✓		
system state validation																✓		
session management				✓	✓					✓	✓					✓		
state synchronization																✓		

Authorization is also described by most papers as a security objective, requirement, or goal, and as a security feature that realizes these. Further, it is defined as a means to achieve access control, with an access source, access target, and actions that are permitted to be executed (Busch and Wirsing 2015). In our example's EHRS, after a user has authenticated their identity, authorization determines what actions this user can perform in the system, i.e., viewing or writing medical records. *Access quota limitation* refers to limiting the usage of resources so that high-priority actions that could be security sensitive are not delayed by other relatively low-priority tasks.

Several schemes can be implemented to assign permissions to users for different purposes. In *attribute-based access control* (ABAC), the access is determined based on attributes such as requested operation, request parameters, or environmental attributes (Chung et al. 2019). In the EHRS from our example, an attribute decides whether a doctor is designated to a patient. *Discretionary access control* (DAC) is where the resources are restricted based on the identity of the users. DAC has complete trust in the users (IBM 2023a). On the contrary, *mandatory access control* (MAC) grants access to resources based on clearance of users, or a predefined hierarchy (IBM 2023c). *Lattice-based access control* determines access to the resources based on a hierarchical lattice structure that represents possible interaction between the resources and the users. This lattice structure is created based on the security levels of the resources and the users (Denning 1976). *Location-based access control*, as the name indicates controls the access based on the location of the user (Ardagna et al. 2009). *Role-based access control* restricts access based on the roles assigned to the users (Ferraiolo and Kuhn 2009), such as doctors or hospital staff. *Rule-based access control* is established based on a predefined set of access rules. *Timed access control* enforces permission or access to resources based on time parameters such as schedule or duration of access. *State-based access control* introduces more fine-grained access decisions than a simple "allow" or "deny" (Kamra and Bertino 2010). For example, "request suspension" is a decision that requires a further negotiation process before deciding whether to grant access (Bertino et al. 2011). *Application mode-based access control* is a special case of state-based access control (Bosch 2000). In summary, all different

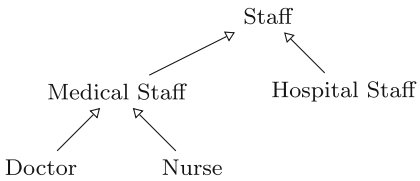

Fig. 7 Simplified role hierarchy in the exemplary EHRS

access control schemes are implemented such that some property of the access source and/or access target is checked against specific requirements. The EHRS from our example uses a simplified role- and attribute-based access control scheme – a common combination in the literature (Jin et al. 2012; Ahmadian et al. 2017). Since multiple roles share permissions, a role hierarchy is implemented in which permissions for a role are inherited from a parent role (see Fig. 7). For example, a doctor should have the same permissions as the medical staff. Therefore, in the role hierarchy, the role *doctor* should inherit all permissions from the role *medical staff*, i.e., writing medical records, and extend it with additional permissions such as writing diagnoses. Since a patient can choose a doctor to be a designated doctor, an attribute denotes whether they have elevated rights over the access to the patient's data.

4.1.2 Cryptography

The feature *Cryptography* aims to ensure secure communication in the presence of adversaries (Rivest 1990). The goal is to prevent unauthorized entities from reading a message by binding a key to the message. A key is a secret consisting of a string of symbols that is used by an algorithm to encrypt or decrypt a message.

Except for *P12*, all investigated papers list cryptography as a security feature, which should be considered when implementing software systems. In our taxonomy, we separated cryptography into five sub-categories that focus on different aspects of cryptography: *encryption*, *cryptographic hashing*, *key management*, *signature*, and *steganography* (see Fig. 8).

Encryption focuses on features for encoding messages in a way that only authorized entity access is able to access its content and protects it from unauthorized modification. Algorithms used for encryption and decryption purposes are called ciphers and can be divided into groups of *stream ciphers* (Jiao et al. 2020) and *block ciphers* (Robshaw 1995). Rivest Cipher 4 (RC4) is one of the most widely used stream ciphers included in various protocols such as TLS. In contrast, block ciphers encrypt a group of plaintext symbols as one ciphertext block. The Data Encryption Standard (DES), triple DES (3DES), and the Advanced Encryption Standard (AES) are well-known block ciphers, which are used in modern software systems.

In an EHRS such as the one from our example, patient data must be encrypted before it is stored within the system using any of the described ciphers, in case a third party is able to intercept data transmitted to or within the system. Any cipher must define a key that is shared among the multiple parties involved in an encrypted communication. In *symmetric key cryptography* (Bokhari and Shallal 2016), a single key is used for both encryption and decryption, while *asymmetric key cryptography* defines a public key, which is used to encrypt a message, and a private key, which is exchanged between the communication parties and used to decrypt the received message (Yassein et al. 2017). A *hybrid cryptosystem* combines the two approaches by using asymmetric key cryptography to encrypt a key from symmetric

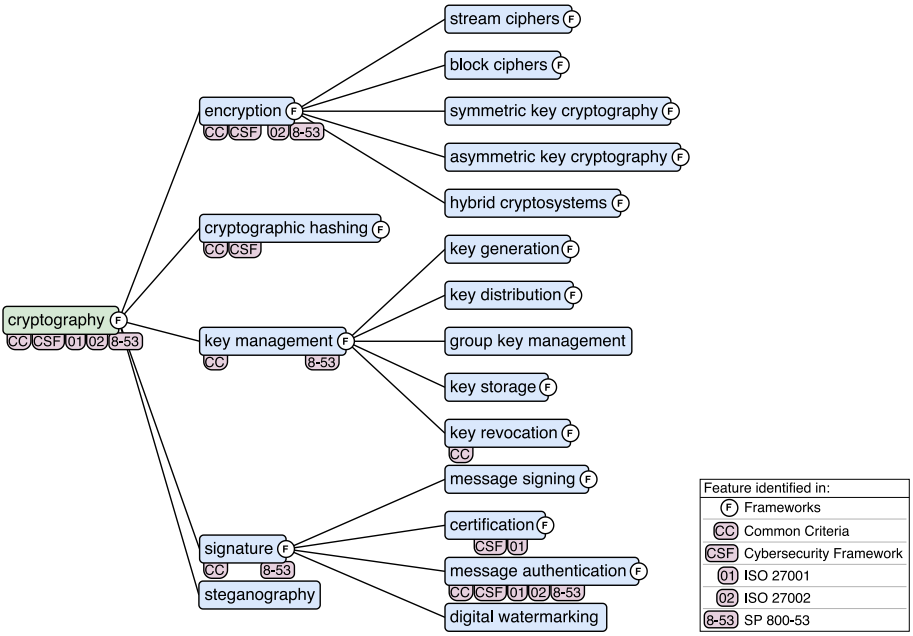

Fig. 8 Sub-features of the top-level security feature *cryptography*

key cryptography (Dent 2004). While a symmetric key algorithm can either use a block cipher or a stream cipher to handle encryption or decryption, asymmetric cryptosystems rely on specific algorithms such as the Rivest Shamir Adleman (RSA) or the Diffie-Hellman exchange method to securely negotiate keys over a public transmission channel (Bhanot and Hans 2015).

Cryptographic hashing focuses on ensuring that data has not been modified, e.g., a message exchanged between a sender and a receiver, without comparing the entire data. To this end, cryptographic hashing functions irreversibly transform data of arbitrary length into a fixed-length output of enciphered text (Busch and Wirsing 2015). Using cryptographic hashing for medical records in the EHRS from our example, ensures that malicious modifications to them can be detected. For the same input, the enciphered output is always identical and, therefore, allows a comparison of the calculated value before and after transmission of a message, while also ensuring the confidentiality of the data by making it unreadable for an attacker. In our exemplary EHRS, passwords in plain text are a major risk to the confidentiality of user data in the system. Therefore, passwords for each user are stored in the database after state-of-the-art cryptographic hashing is applied. In case of an incident in which passwords are stolen by an attacker, they are unable to gain access to the stored user accounts, since they can not decipher the hash.

Key management is essential since cryptographic operations rely on secure and confidential keys (Rana et al. 2023). First, *key generation* must be performed in a secure way, which requires the usage of securely generated random numbers. Second, to be able to encrypt and decrypt messages, the communicating parties must exchange a key via a *key distribution* scheme (with *group key management* as a special form of key distribution). Once a key has

been negotiated, only authorized users should be allowed access to it, which can be ensured by using a *key storage* method. Finally, *key revocation* is used to invalidate a key once it is not required anymore, e.g., after a certain timeframe has passed or a criterion has been met.

Signature is used to verify the authenticity of a message by binding the identity of the sender to the sent message (Katz 2010). In this process, *message signing* is used to create and bind a signature to a message by associating it with the private key of the sender. For instance, medical records are signed with a key of a doctor to establish authenticity of the message in our exemplary EHRS. Similarly, a *certification* can be realized by a third party to show that a key can be trusted. Thereafter, the receiver can use *message authentication* to verify the origin and that the received message has not been tampered with in transit. Here, the corresponding public key is then used to verify the private key bound to the message. *Digital watermarking* is a method to attach a non-removable signature to data to ensure its origin cannot be tampered with by a third party.

Steganography can be used to hide a message within another, potentially without the use of an encryption algorithm (Kour and Verma 2014). While there is a chance for an unauthorized entity to access the message, the idea is that unknowing entities are not able to notice that secret information is hidden within the message. One such method hides a message within an image in a way that it cannot be perceived by humans.

4.1.3 Security monitoring

The top-level feature *security monitoring* (Fig. 9) describes features for monitoring properties of software systems that can indicate the state of security or possible security issues. For example, monitoring network traffic can be used to detect intrusions or other issues (Ghafir et al. 2016). In general, monitoring a software system can reveal attempted attacks and help prevent their success (McGraw 2004). The feature contains *automated response*, *history maintenance*, and *logging* as sub-features. It is covered by ten of the ontologies (see Table 2).

Automated response refers to responding to incidents that happen in a software system that can lead to potential security violations. If automated responses independent of human interaction are implemented, the response time to security incidents can be reduced.

Logging, while not used as a feature to prevent attacks, can identify and trace back anomalies, such as attacks, within a system. Whenever a user such as a doctor writes or saves data to the EHRS from our example, the event is logged to a log file, containing the user, time, and action that was performed. This assures the accountability of certain actions, and helps in reasoning about incidents that may occur. Additionally, several considerations regarding the security of the content of the logs must be taken. As such, the secure logging pattern intends to prevent an attacker from gathering sensitive data about a system from its logs (Dougherty

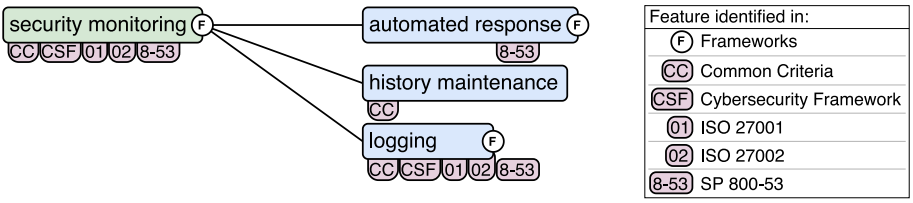

Fig. 9 Sub-features of the top-level security feature *security monitoring*

et al. 2009). In our exemplary EHRS, all logs are encrypted and access to them is restricted to specific users. Past work presents logging as a mechanism for providing non-repudiation and ensuring system and data integrity (*P4, P8, P9, P13, P16, P17*), incident management (*P1, P16*), and intrusion detection and prevention (*P10, P14*).

History maintenance preserves the user/system activity logs, enabling the lookup and identification of wrongdoers or unwanted incidents in cases of a security breach. The storing of relevant information has to be implemented in software systems to allow such investigations after an issue has been detected. In the EHRS example, changes resulting from system interactions are retained so that, for instance, previous versions of physician reports can be restored.

4.1.4 Secure data handling

This top-level feature is mentioned in 8 out of the 18 papers that we examined. The feature *Secure data handling* covers security features that deal with validation, sanitization, control, and secure storage of the data that is handled in a software system (see Fig. 10). Since data management is a core feature within any EHRS, secure data handling plays a crucial role.

Data validation is characterized by two sub-features, *input validation* and *output validation*. For example, input validation ensures that entered data complies with a valid data format and does not contain malicious data, such as scripts. To this end, *blacklisting* or *whitelisting* can additionally be used to restrict or trust sources from which data can be introduced into the system.

Download verification of data that is obtained from external sources can assert that—in addition to not containing any malicious scripts or similar—has not been tampered with and contains the expected content.

Data sanitization, which involves *input and output sanitization* (e.g., escaping the user-provided inputs before using them in any kind of database query in the EHRS from our example), and *parameterized prepared statements* (e.g., pre-compiling an SQL statement before patient data is accessed) form another branch of secure data handling. These features need to be implemented to ensure that no malicious inputs of a potential attacker are permitted and that no sensitive data such as passwords are leaked, i.e., they can mitigate attacks such as SQL injections (Shar and Tan 2013).

Retention control is another feature under secure data handling that deals with the secure management of data that is no longer needed for any operations but is still maintained in

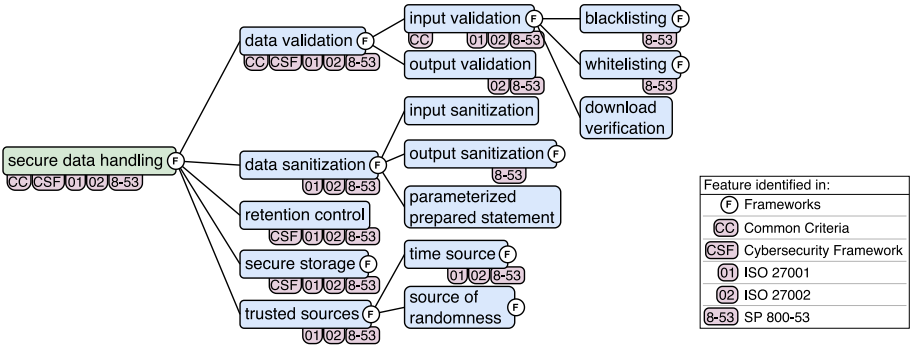

Fig. 10 Sub-features of the security feature *Secure Data handling*

the system. For instance, personal patient data related to billing will be deleted from our exemplary EHRS after a certain timeframe has passed. Specifically, it deals with the duration or other indicator for when unused data should be deleted or under which circumstances this should not be the case. The underlying principle is to reduce the attack surface of a system by minimizing the amount of sensitive data that could be accessed by an unauthorized user or system component if compromised.

Secure storage describes storing user and other data in a way that keeps it from being accessed by unauthorized users or software components, preserving the confidentiality and integrity of the stored data (Löhr et al. 2010). In the EHRS from our example, cryptographic keys for medical records are not stored in the same database as the medical records themselves. For instance, *P11* introduces an encrypted storage feature to securely store data.

Trusted sources involve the secure generation of timestamps (*time source*) and random numbers (*randomness*). A trusted source of time ensures, e.g., that logs can be trusted and used for investigations after a security incident, also between distributed systems that share a trusted source of time. A trusted source of randomness is vital for many cryptographic operations, e.g., as seeds for encryption protocols or for key generation (Schindler 2009).

4.1.5 System state protection

The top-level security feature *system state protection* (Fig. 11) describes security features that ensure that the system's operational state is not compromised and that it conforms to defined requirements. It is mentioned in six of the analyzed ontologies. In our exemplary EHRS, system state protection is implemented by requiring a doctor to read a patient's data file before they are allowed to prescribe medication via the system to avoid mistreatment.

Resource management refers to implementing control mechanisms for the allocation and access of resources based on a priority level to ensure availability. As such, resources must be managed to warrant that no attacker can take down a software system by reserving large amounts of resources, e.g., via DDoS attacks (Mirkovic and Reiher 2004), as occurred in the incident at the hospital (BBC 2020) described in Section 2.1.

System state validation can ensure that the system is in a secure state, i.e., that it has not been compromised and that its operational state is correct and secure according to predetermined rules. This feature is especially important after events such as the boot-up of the system or recovery after an incident.

Session management describes security features mainly to prevent attacks on web applications. Four security features to mitigate attacks related to sessions are presented here, *replay attack prevention*, *session fixation protection* (Johns et al. 2011), *session takeover prevention* (Baitha and Vinod 2018), and *session timeout*. Session management plays a critical role in our exemplary EHRS by utilizing session timeouts, which closes a session after a period of inactivity, preventing other users in the hospital to gain access to the session.

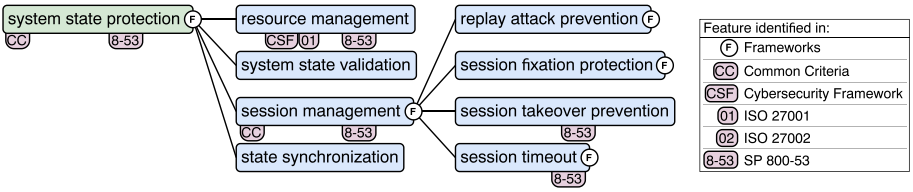

Fig. 11 Sub-features of the top-level security feature *system state protection*

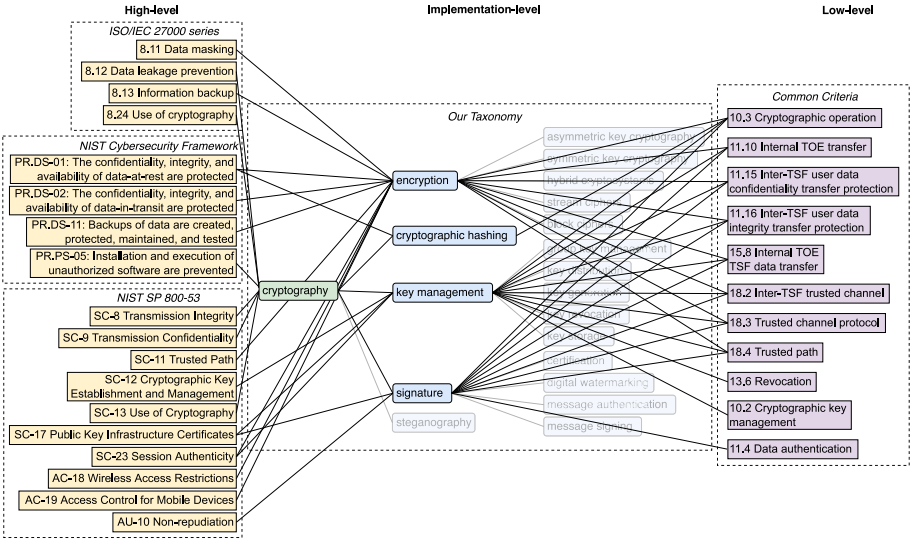

Fig. 12 Partial excerpt of the mapping from high-level security standards such as the ISO/IEC 27000 series (left) and from the detailed Common Criteria (CC) (right) to implementation-level security features in our taxonomy (middle). The taxonomy presents security features that can provide the security requirements specified by the high-level standards and generalizes specific low-level details in the CC to actionable advice for developers

State synchronization ensures that the states in a system are consistent and synchronized between distributed functions. This prevents attacks that exploit differences in states between system components.

4.2 Relation to security standards

Following the methodology described in Section 3.1.3, we mapped our taxonomy to security standards, thereby also validating and adapting it. The mapping and the presented descriptions of the security features can guide developers in adhering to the security requirements of the high-level standards. In particular, starting from the high-level security standards, developers can use the mappings to select suitable functional security features to realize from our taxonomy. For the concrete realization, they can then follow the mappings into the relevant detailed aspects of the CC. In particular, Part 2 of the CC[1] contains detailed descriptions of implementation-level security features that fit the scope of our taxonomy, but the other standards also contain functional security features for which mappings are expected.

4.2.1 Example mapping

As an illustrative example for the mapping, Figure 12 shows the mapping of the top-level security feature cryptography and its sub-features to the security standards and guidelines used as comparators. For brevity, we focus on this one feature to illustrate the mapping between our taxonomy and the standards and guidelines. The complete mapping is made publicly available in our online Replication Package (2025). As shown on the left-hand side

[1] https://commoncriteriaportal.org/files/ccfiles/CC2022PART2R1.pdf

in Fig. 12, multiple parts of the ISO/IEC 27000 series, as well as the NIST Cybersecurity Framework and the NIST SP 800-53, relate to security features in our taxonomy. On the right-hand side of Fig. 12, the parts of the CC are shown that relate to security features in our taxonomy. Here, the descriptions in the CC are specific descriptions of certain technologies or techniques to realize the security features they are mapped to. To this end, the taxonomy provides a more general description of the content of the CC.

4.2.2 Mapping and refining of the taxonomy

In this section, we present the analyzed security standards and describe the process of mapping our taxonomy to the standards and our changes to the taxonomy. Slight adjustments were made, e.g., to render the taxonomy more generally applicable when we found that the standards described certain security features in a broader scope than their representation in our taxonomy. Other changes were the addition of further security features that were not contained in the literature and some adjustments to the structure of the taxonomy. These changes are already considered in the description of the taxonomy presented in Section 4.1, i.e., the following describes the last refinement steps we had undertaken to reach the final taxonomy presented above. Table 3 shows the overlap between the final taxonomy and the security standards. Overall, nine security features were identified across the standards that were not contained in the taxonomy prior to this analysis. We have adjusted our final taxonomy accordingly. Out of the 68 security features in the taxonomy, 45 were identified in at least one of the standards, whereas 23 are not mentioned in any of them. The biggest overlap between an individual standard and the taxonomy was observed for the SP 800-53, with 26 security features identified in it that are in the taxonomy. The ISO 27002 showed the smallest overlap with only 12 security features from the taxonomy identified in it. The overlaps and performed changes are described in detail in the following.

Common Criteria (CC) The CC presents details on low-level security features. It offers a basis for certifying the security of IT products by listing security properties that need to be fulfilled or for which assurance needs to be provided. In contrast to the high-level descriptions that are found in many standards, the content of the CC is largely on the implementation level. Often, specific technologies for achieving a certain security functionality are presented, lacking generality.

The analysis of the CC revealed that 55 of its chapters describe implementation-level security features. Of these, 39 (71%) directly mention one or more of 16 security features

Table 3 Overlap between security features of standards and taxonomy, showing how many are covered by the taxonomy and how many are missing in the taxonomy and standards

	Security Features		
Standard	**in Standard**	**not in Taxonomy**	**not in Standard**
CC	25 (16*)	0 (9*)	43
27001	13	0	55
27002	12	0	56
SP800-53	26	0	42
CSF	14	0	54
Overall	45 (36*)	0 (9*)	23

* Prior to adjusting the taxonomy based on the CC

already contained in the taxonomy. A mapping between two further chapters and the taxonomy could be achieved by rephrasing the previous feature *escaping user-supplied input* found in the literature to the more general *input sanitization* and *output sanitization*, and rephrasing *log monitoring* to the more general *security monitoring*. After these changes, we checked the ontologies from the literature that were used to create the taxonomy again and verified that these less specific feature names still fit the descriptions in the literature, which was the case.

The remaining 14 chapters from the CC required the addition of new security features to our taxonomy because they described security features that were too dissimilar to the features already contained in our taxonomy. Consequently, we added the nine security features *trusted sources* of *time* and *randomness*, *secure storage*, *replay attack prevention*, *resource management*, *retention control*, *system state protection*, *system state validation*, and *state synchronization* to the taxonomy (note that multiple chapters in the CC can relate to the same security feature, therefore the disparity between 14 previously un-mapped chapters and the addition of only nine features). To validate the updated taxonomy's accordance with the literature, we looked for descriptions of the newly added security features in the ontologies that were the initial sources. We identified references to all nine of them. In the ontologies, the descriptions were less concrete than in the CC (for example, paper *P10* presents *system availability* in combination with *DDoS prevention*, which indicates the newly added security feature *resource management*), which is why we did not initially include them in the taxonomy.

The addition of further security features made some adjustments to the structure of the taxonomy necessary to achieve a more coherent grouping. The new structure better reflects the level of granularity, meaning that now, when comparing two security features in the taxonomy, the lower-level one (i.e., the one more to the right in Fig. 12) generally addresses a more specific security issue or requirement than the higher-level one (i.e., the one more to the left in Fig. 12). One of two changes to the structure concerned the feature *secure data handling*. Previously, the feature *data validation* had been a top-level feature, which we changed to *secure data handling* being the top-level feature. These changes do not contradict the ontologies we used as initial sources from the literature, since they have no strict hierarchy.

A second adjustment to the structure of the taxonomy was prompted by the addition of the feature *system state protection*. Previously, the feature *session management* had been a top-level feature. *System state protection* is an important security feature on a similar level of granularity as the top-level features in our taxonomy. Even though it is presented as a security feature in the CC, it is not mentioned in any of the ontologies we examined, which shows a gap in the literature. We decided that the most coherent structure would be to add it as a top-level feature and organize the features connected to it into the final form shown in Fig. 11.

In summary, the functional security features described in the CC could be mapped very well to the taxonomy derived from the literature. The majority could be mapped directly. For the others, we performed slight adjustments to the taxonomy by adding further security features, rephrasing existing ones, and changing the structure.
The taxonomy was improved by these changes and in its final form (shown in Fig. 5) not only represents the literature but also the CC.

ISO/IEC 27000 family (27001 and 27002) This group contains multiple standards that apply to software security in general or to software security of specific domains. Two standards are

specifically related to our work, the *ISO/IEC 27001* and the *ISO/IEC 27002*, which are the two major standards in the family. ISO/IEC 27001 describes requirements for establishing, maintaining, and improving an Information Security Management System (ISMS). It presents controls and objectives that organizations might adopt, based on their unique risk landscape. This standard covers organizational, people, physical, and technological controls. It primarily offers high-level policies and requirements, making it challenging to identify specific security features that should be implemented. ISO/IEC 27002 provides guidance and reference for implementing security features to manage information security risks in an ISMS based on ISO/IEC 27001. It is more explicit than the ISO/IEC 27001 on implementation details of security features. However, most of the standard still consists of high-level descriptions rather than actionable guidance for developers. In addition, the sheer volume and depth of ISO/IEC 27002 makes identifying implementation-specific features a tedious task, further emphasizing the utility of the taxonomy derived in this paper.

Analyzing the technological controls presented in the standards revealed that many of their high-level descriptions can be mapped to the implementation-level security features in our taxonomy. Here, the security features are a way of realizing the technological controls in the standards. Out of the 34 technological controls that are presented in ISO/IEC 27001 and specified further in ISO/IEC 27002, 14 (41%; 13 from ISO/IEC 27001 and 12 from ISO/IEC 27002) can be mapped in this way to 15 of the security features in our taxonomy. The remaining 20 technological controls can not be realized with implementation-level security features but instead, describe organizational and procedural requirements. For example, technological controls require that the organization implements hardware redundancy (control 8.14), secure coding principles (control 8.28), or change management processes (control 8.32). We mapped the 14 technological controls that can be mapped to the taxonomy to the 15 security features *access control*, *authentication*, *data validation*, *encryption*, *input validation*, *logging*, *certification*, *resource management*, *retention control*, *secure storage*, *cryptography*, *data sanitization*, *security monitoring*, *trusted source of time*, and *output validation*. For some mappings, the security features are given as examples of how a technological control can be realized, while others are not explicitly named but the technological controls fit the security features' descriptions (for example, the technological control *Information stored in information systems, devices or in any other storage media shall be deleted when no longer required* is mapped to the security feature *retention control*).

Overall, no changes to the taxonomy were required to map all implementation-level security features found in the ISO/IEC 27000 family of standards to our taxonomy.

NIST SP 800-53 This standard presents a wide variety of security (and privacy) requirements and related controls. The descriptions refer to organizational and procedural actions for the most part, only occasionally mentioning implementation-level security features. In general, the standard calls for an "organization-wide process to manage risk", hence not focusing on technical controls alone. The description of the faced threats and attacks as "hostile attacks, human errors, natural disasters, structural failures, foreign intelligence entities, and privacy risks" further shows the scope of the document and the reason why implementation-level security features are scarce.

Nevertheless, similarly to the ISO/IEC 27000 family, large parts of the NIST SP 800-53 could be mapped to our taxonomy. Out of the 78 *technical controls* that the standard presents,

40 (51%) are related to one or more security feature(s) in our taxonomy, meaning that the security features allow the realization of the technical controls. The remaining 38 technical controls do not require implementation-level security features. Instead, they describe common software security practices (e.g., *principle of least privilege* (technical control AC-6) or *separation of duties* (technical control AC-5)), user-oriented features (e.g., the display of privacy and security notices (technical control AC-8) or display of information about the last logon (technical control AC-9)), or other information that is non-functional, too specific, or not on the implementation-level.

All the 40 technical controls presented in the NIST SP 800-53 that describe implementation-level security features could be mapped to 26 features in our taxonomy without changes.

NIST Cybersecurity Framework 2.0 (CSF) This resource offers high-level guidelines for organizations to pinpoint risks and threats, along with recommended processes to address them. It emphasizes the importance of implementing security features as protective measures against potential threats. However, due to its broad scope (for instance, the detection of and recovery from attacks are also covered), the document generally lacks in-depth details on specific implementation-level security features.

In total, the document is structured into 23 *categories* that describe measures to protect software systems. Out of these, 14 categories (61%) could be mapped to 14 security features in our taxonomy. The other nine categories are non-functional or beyond the scope of our taxonomy for other reasons (e.g., PR.AT-01 and PR.AT-02 are concerned with awareness and training of users, PR.PS-02 and PR.PS-03 ask for the consideration of risks in software and hardware maintenance, replacement, and removal, and PR.IR-02 relates to environmental threats).

All 14 implementation-level security features identified in the NIST Cybersecurity Framework 2.0 could be mapped to our taxonomy without any changes.

Figure 13 visualizes the overlap between the standards and our (initial and final) taxonomy in terms of the number of security features (see Table 3). As shown, the initial taxonomy covered all 32 security features mentioned in the four analyzed standards (summarized as *Other standards*) and extended this set with 27 further features found in the academic literature. The overlap between the initial taxonomy and the CC was 16 security features, 12 of which are also contained in the *Other* standards. The CC contained 9 additional features that were not part of the initial taxonomy. After extending the taxonomy with the 9 features from the CC, the final taxonomy contains 68 functional security features. It covers all 32 features found in the four analyzed standards and all 25 features identified in the CC. Additionally, the final taxonomy contains 23 features that resulted from our review of the literature and that are not present in any of the standards or the CC. Based on this presentation and the above descriptions of the mappings between the security standards and guidelines to our taxonomy, we can answer RQ1 as follows.

RQ1: We collected 68 implementation-level security features from the literature and security standards. We identified five of them as top-level security features to provide the structure for the taxonomy: *access control*, *cryptography*, *secure data handling*, *security monitoring*, and *system state protection*. The taxonomy presented in Fig. 5 provides all security features.

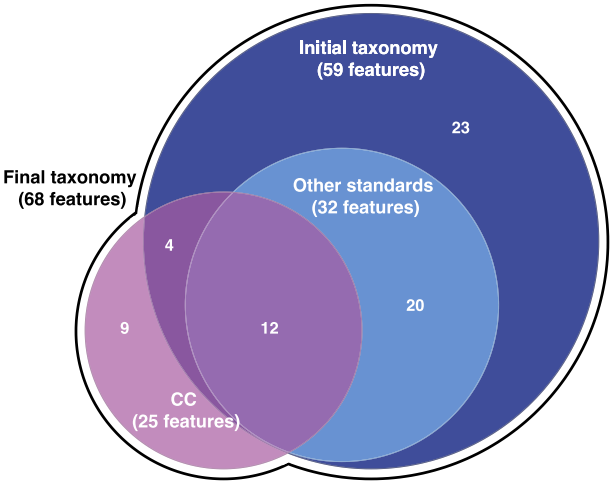

Fig. 13 Venn-diagram presenting the overlap between the analyzed standards and the (initial and final) taxonomy concerning security features identified in them

5 Security features provided by security frameworks (*RQ2*)

We now present our investigation of the security features supported by security frameworks. We discuss the differences between the security features presented in the literature, as documented in our security feature taxonomy (i.e., Section 4), and those provided to developers by commonly discussed security frameworks.

5.1 Identified security frameworks

We selected 21 security frameworks via our search on Stack Overflow and Reddit. Table 4 shows the selected frameworks, sorted by the number of threads on Stack Overflow and Reddit in which each framework was mentioned. Some frameworks in the table are positioned based on the total number of threads within their parent frameworks, such as ASP.NET, which is a component of the larger .NET framework. While some framework may be considered outdated when it has not received any updates since 2020, they may still be used by developers, e.g., JGuard was discussed four years after the release of its last stable version, the Java Security Manager is deprecated but still part of maintained JDK versions, and some frameworks still show downloads at the time of writing.

In total, we identified 44 security features offered to developers by the selected security frameworks. Terms found in the respective reference guides and documentation were grouped according to our methodology in Section 3.2.2. The grouping in Table 5 is based on the structure of our taxonomy which denotes security features present in each framework. Some of the frameworks can also be used for different purposes than utilizing them within a software system, such as using them as a standalone application for encrypting a file. However, we only consider the security features offered by security frameworks in a scenario in which developers implement software systems by embedding them at the code-level. Many security frameworks are offered for different programming languages or focus on specific security features, which makes it difficult to compare them. However, they may offer similar security features, which may differ in their specific implementation.

Table 4 List of selected security frameworks. # = Number of threads. SO = Stack Overflow. M = Maintained

ID	Security framework	URL	# SO	# Reddit	M
01	Spring Security	spring.io/projects/spring-security	91	2	Yes
02	Apple Security Framework	developer.apple.com/documentation/security	37	0	Yes
03	Apache Shiro	shiro.apache.org	25	0	Yes
04	JAAS	docs.oracle.com/javase/8/docs	12	0	Yes
05	Java EE	oracle.com/java/technologies/java-ee-glance.html	2	0	Yes
06	Java Security Manager	docs.oracle.com/javase/tutorial	2	0	No
07	OpenSSL	openssl.org	11	2	Yes
08	Windows Identity Foundation	microsoft.com/en-us/download/details.aspx?id=17331	2	0	Yes
09	ASP.Net Membership Provider	learn.microsoft.com	6	0	Yes
10	ASP.Net Role Provider	learn.microsoft.com	3	0	Yes
11	OWASP ESAPI	owasp.org/www-project-enterprise-security-api	5	0	Yes
12	JBoss Seam Security	docs.jboss.org/seam/3/security	4	0	No
13	Passport	passportjs.org	3	0	Yes
14	Play Framework Secure Module	playframework.com/documentation/1.2.5/secure	2	1	Yes
15	OACC	oaccframework.org	2	0	No
16	JGuard	jguard.xwiki.com	2	0	No
17	Bouncy Castle	bouncycastle.org	2	0	Yes
18	Endpoint Security Framework	developer.apple.com/documentation/endpointsecurity	2	0	Yes
19	EveryAuth	github.com/bnoguchi/everyauth	2	0	No
20	PicketBox	picketbox.jboss.org	2	0	No
21	Sureness	usthe.com/sureness	2	0	Yes

In what follows, we summarize the investigated security frameworks, highlighting their coverage in different areas of security. We found that the security frameworks can be grouped into two categories that target access control and cryptography. Besides, some security frameworks offer a wide range of security features, and can, therefore, be classified as multi-purpose frameworks.

Access Control Frameworks Based on their offered security features, 15 of the selected security frameworks can be mainly used to implement and manage authentication and authorization. Among them, we identified two authentication and authorization middlewares (Passport and EveryAuth) for node.js. Three modules of the Java standard library implement authentication and authorization (JAAS), enterprise software utilities (Java EE), and security policy enforcement (Java Security Manager). The Windows .Net Identity Foundation is a security framework for facilitating user authentication in software systems. JGuard is a security framework based on JAAS for solving access control problems for web applications.

Table 5 List of security features provided by security frameworks

	01 Spring Security	02 Apple Security Framework	03 Apache Shiro	04 JAAS	05 Java EE / Jakarta EE	06 Java Security Manager	07 OpenSSL	08 Windows .Net Identity Foundation	09 ASP.Net Membership Provider	10 ASP.Net Role Provider	11 OWASP ESAPI	12 JBoss Seam Security	13 Passport	14 Play Framework Secure Module	15 OACC	16 JGuard	17 Bouncy Castle	18 Endpoint Security Framework	19 EveryAuth	20 PicketBox	21 Sureness	
access control																						
authentication	✓	✓	✓	✓	✓				✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	76.2%
authorization	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	85.7%
cryptography																						
cryptographic hashing	✓	✓	✓		✓		✓				✓	✓				✓	✓					47.6%
encryption	✓	✓	✓				✓		✓		✓				✓		✓					38.1%
key management	✓	✓					✓										✓					19.0%
signature	✓	✓					✓										✓					19.0%
steganography																						0.0%
security monitoring																						
automated response						✓																4.8%
history maintenance																						0.0%
logging	✓						✓				✓	✓						✓	✓	✓		28.6%
system state protection																						
resource management																						0.0%
system state validation																						0.0%
session management	✓	✓	✓		✓						✓	✓	✓		✓				✓			38.1%
state synchronization																						0.0%
secure data handling																						
data validation											✓											4.8%
data sanitization											✓											4.8%
retention control																						0.0%
secure storage		✓													✓							9.5%
trusted sources		✓	✓				✓				✓						✓					19.0%

Similarly, JBoss Seam Security and OACC are access control frameworks aiming to provide general functionalities to manage and enforce access control policies. While the Play Framework Secure Module handles authentication and authorization, in ASP.NET, these features are split among Membership Provider and Role Provider. The Endpoint Security Framework offered by Apple can be used to monitor and authorize system events. Finally, the framework Sureness focuses on securing REST APIs.

Cryptography Frameworks OpenSSL and Bouncy Castle offer a large range of *encryption*, *key management*, *hashing*, and *signature* features. While Bouncy Castle is purely a cryptographic library, OpenSSL uses the cryptographic library *libcrypto* for implementing cryptographic features. Cryptographic features are offered by 9 other frameworks as well (Spring Security, Security Framework, Apache Shiro, Java EE, ASP.NET, OWASP ESAPI, JBoss Seam Security, OACC, JGuard).

Multi-Purpose Frameworks Spring Security, Apple Security Framework, Apache Shiro, and OWASP ESAPI offer a large range of security features from our taxonomy, covering both, access control and cryptography features, as well as additional ones such as session management. Notably, OWASP ESAPI additionally provides most features for data handling, such as *data validation*, *data sanitization*, and *trusted sources*.

5.2 Provided security features and relation to the security taxonomy

To investigate the relationship between the functional security features captured in the taxonomy and those provided to developers, we mapped the frameworks' features to the taxonomy (recall, that we marked them with "F" in Figs. 6, 8, 9, 10, and 11).

5.2.1 Provided security features

In the following, we present the identified security features in the order of the taxonomy's top-level security features, as shown in Fig. 5.

Access Control. As shown in Table 5, all frameworks except OpenSSL provide features to realize some kind of access control. Sixteen of the 21 frameworks offer features to realize authentication. However, from the security features in Fig. 6, *multifactor authentication* is not provided by any framework. All frameworks use credentials for authentication but also offer authentication via certificates (Spring Security). Additionally, Spring Security and Passport provide authentication features via single sign-on. The feature one-time-password is only provided by PicketBox.

```
1  public class LoginServlet extends HttpServlet {
2    // The GET method is called when a user clicks submit on the login
       ↪ page
3    protected void doGet(HttpServletRequest request,
       ↪ HttpServletResponse response) throws ServletException,
       ↪ IOException {
4      // Get user and password from HTTP response
5      final var user = new SimplePrincipal(request.getParameter("user")
       ↪ );
6      final var password = request.getParameter("password").toCharArray
       ↪ ();
7
8      /* Creating a LoginContext automatically loads the authentication
       ↪  mechanism registered
9      with JAAS. User and password are provided via a CallbackHandler.
       ↪ */
10     var context = new LoginContext("ehrs", new CallbackHandler(user,
       ↪ password){
11       public void handle(Callback[] callbacks){
12         // Callbacks used for authentication are provided with user
       ↪ and password
13         for(var callback : callbacks) {
14           if(callback instanceof NameCallback nameCallback) {
15             nameCallback.setName(user);
16           } else if(callback instanceof PasswordCallback
       ↪ paswordCallback) {
17             passwordCallback.setPassword(password);
18           }
19         }
20       }
21     });
22     try {
23       context.login(); // Authenticating the user throws a
       ↪ LoginException if it fails
24     } catch (LoginException e) {
25       // In case of a failed login, a simple failure page is returned
26       response.getWriter().println("<b>authenticatation failed!</b>")
       ↪ ;
27     }
28   }
29 }
```

Listing 1: Servlet of our exemplary EHRS login page that implements authentication of a user using JAAS

In our running example, the authentication of users may be realized using JAAS. Listing 1 shows a possible implementation of this security feature. Our exemplary EHRS would be implemented using Java Server Pages (JSP). Whenever a user clicks the login button on the login page, the Servlet shown in Listing 1 is called. In this case, the `doGet` method is called and handed over an HTTP request containing the data entered into a *user name* and *password* field on the login page. This information is retrieved from the request (lines 5 and 6), and then, the JAAS login is instantiated in lines 9–20. JAAS automatically loads an authentication mechanism registered with JAAS when instantiating `LoginContext`. The user name and password are handed over via callbacks, which in this case are provided with the values retrieved from the HTTP request in lines 13–18. Besides setting login information in code, JAAS supports various callbacks to provide data, e.g., callbacks that show pop-ups to users in a desktop application. After instantiating the login context, the provided login information is validated in line 23 by calling `login()`. All of the Java security frameworks examined are compliant with JAAS and extend it by, for example, registering authentication mechanisms or providing easier-to-use wrappers for specific usage scenarios such as web applications.

```
1  @RestController
2  @RequestMapping("/patient-info")
3  public class PatientInfoController {
4
5      private String patientInfo = "Sensitive Patient Information";
6
7      @PreAuthorize("hasRole('ROLE_DOCTOR') && hasPermission(#patient, '
       ↪ designated')")
8      @GetMapping
9      public String getPatientInfo() {
10         return patientInfo;
11     }
12
13     @PreAuthorize("hasRole('ROLE_DOCTOR')")
14     @PostMapping
15     public String updatePatientInfo(@RequestBody String newInfo) {
16         patientInfo += newInfo;
17         return "Patient information updated successfully.";
18     }
19 }
```

Listing 2: An example of implementing role- and attribute-based access control with the Spring Security framework.

After a user has been authenticated, it is usually decided whether the user is allowed to perform specific activities. Even though authorization is one of the most prevalent security features realized in 18 of the 21 frameworks, only *attribute-based* and *role-based access control* are offered. Figure 6 emphasizes that frameworks are missing most sub-level features of the authorization feature. Listing 2 shows an example of enforcing access control in our exemplary EHRS using Spring Security. To implement the access control scheme of the EHRS, a combination of role-based and attribute-based access control would be implemented. First, patient information from the endpoint `/patient-info` can only be accessed by users with the role `ROLE_DOCTOR`. (see lines 7 and 13). However, to be able to read the patient information (`getPatientInfo()`), this doctor must also be a designated doctor for the patient, which is expressed as an attribute that is implemented using the 'designated' permission for the corresponding patient (see line 7).

```java
public byte[] encryptPatientData(String data) {
  // Pasword and salt for key derivation
String password = "...";  // some random password
String salt = KeyGenerators.string().generateKey();

 /* Create a password-based encryptor
 using 256 bit AES encryption */
 var aes = Encryptors.stronger(password, salt);

 // Encrypt the patient data
 return aes.encrypt(data.getBytes());
}
```

(a) Encryption using Spring Security

```java
Security.addProvider(new BouncyCastleProvider());

public byte[] encryptPatientData(String data) {
  // Pasword and salt for key derivation
  char[] password = "...";  // some random password
  byte[] salt = ...;

  // Generate AES key
  var keyGen = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
  var spec = new javax.crypto.spec.PBEKeySpec(password, salt, 1024, 256);
  var secretKey = keyGen.generateSecret(spec);

  // Create and initialize cipher
  Cipher cipher = Cipher.getInstance("AES/GCM/PKCS5Padding", "BC");
  cipher.init(Cipher.ENCRYPT_MODE, secretKey);

  // Encrypt the patient data
  return cipher.doFinal(data.getBytes());
}
```

(b) Encryption using Bouncy Castle

Listing 3: Credential-based Encryption using an AES Block Cipher in Spring Security and Bouncy Castle, using the predefined parameters of Spring Security.

Cryptography. In addition to the cryptography libraries OpenSSL and Bouncy Castle, the framework Spring Security and the Apple Security Framework offer the most cryptography features. In total, eleven frameworks offer security features for cryptography. Excluding Bouncy Castle and OpenSSL, half of the frameworks offer *cryptographic hashing* which is designed explicitly for access control or password encryption (Spring Security, Security Framework, Apache Shiro, OWASP ESAPI, JBoss Seam Security, OACC, JGuard). Eight frameworks provide support for *encryption*, offering a diverse selection of cryptographic algorithms. These algorithms span across *symmetric cryptography*, including *block and stream ciphers*, and *asymmetric cryptography*. In addition to enabling the generation and verification of *signatures*, the frameworks Spring Security, Apple's Security Framework, OpenSSL, and Bouncy Castle (19.0%), offer capabilities for *key management*.

An example of how the encryption in EHRS can be implemented using Spring Security is shown in Listing 3a, while the same encryption using Bouncy Castle is shown in Listing 3b for comparison. While Spring Security abstracts most of the configuration details from the user,

therefore allowing limited configuration, Bouncy Castle allows for detailed configuration but is more complicated to use. Both require credentials and a seed for encryption. While Spring Security comes with a utility function to generate the seed (line 4 in Listing 3a), the seed for Bouncy Castle must be generated in handwritten code. In both frameworks, a secret key is generated from a password and salt, but in Spring Security this is hidden from the user. Spring Security provides some standard configurations via the class `Encryptors`, of which we initialize the `stronger` variant in line 8 of Listing 3a. In lines 9 to 15 of Listing 3b, we configure Bouncy Castle in the same way as the selected configuration. First, we generate a secret key in lines 9 to 11, and then we initialize the cipher used for encryption in lines 14 and 15, in both cases using the predefined configuration parameters of Spring Security. Finally, in both Spring Security and Bouncy Castle, the data is encrypted (line 10 in Listing 3a and line 18 in Listing 3b).

The features *steganography*, *group key management*, and *digital watermarking* are not provided by any of the security frameworks considered (see Fig. 8).

Security Monitoring. Seven frameworks offer logging features (Table 5). Although *logging* is not inherently designed as a proactive defense against attacks, it plays a crucial role in identifying anomalies within a system and retroactively tracing back problems of a system. Most frameworks offer some support for integrating external logging frameworks. Finally, Java Security Manager is the only security framework that offers the configuration of *automated responses* to security incidents.

```
1 public class SessionListener implements HttpSessionListener {
2    @Override
3    public void sessionCreated(HttpSessionEvent e) {
4       // Set session timeout in seconds
5       e.getSession().setMaxInactiveInterval(300);
6    }
7 }
```

(a) Session timeouts in Spring Security using the SessionListener

```
1 [main]
2 ...
3 # 300.000 milliseconds = 5 minutes
4 securityManager.sessionManager.globalSessionTimeout = 300000
```

(b) Session timeouts in Apache Shiro configuration file shiro.ini

Listing 4: Setting session timeouts in Spring Security and Apache Shiro

System State Protection. Session management and three of its sub-features are the only *system state protection* features offered by eight of the security frameworks. As such, many of the *system state protection* features rely on the manual implementation of developers rather than the usage of security frameworks. Listing 4a shows an example of how session timeouts can be implemented in the EHRS using Spring Security, which relies on the implementation of a `SessionListener` provided by the Java Standard Library. In contrast, implementing session timeouts in Apache Shiro relies on configuring a session manager provided by the security framework itself as shown in Listing 4b.

```
1 let stat = SecItemAdd(addquery as CFDictionary, nil)
2 guard stat == errSecSuccess else {throw <# error #>}
```

Listing 5: Adding data to a keychain in the Apple Security Framework

Secure Data Handling. Only OWASP ESAPI offers security features for *data validation* and *data sanitization*, including some of its low-level security features (see Fig. 10). Although OWASP ESAPI provides a set of methods for data validation, the capabilities of the offered validations are limited to basic validations and, therefore, require developers to extend these with validation rules tailored to specific security requirements of their applications. Only the Apple Security Framework and OACC offer means to realize secure storage. The former offers a secure storage solution named keychain which assists developers in implementing secure storage, with an example of adding data to it shown in Listing 5. The latter, OACC offers a comparable solution by providing a fully implemented database specifically designed to manage security-related information. This is achieved through the execution of setup scripts tailored to different databases. Four frameworks provide features to create trusted sources, such as random number generators and timestamps. The latter is a foundational security feature that can be used with other features, such as *retention control* or *session takeover prevention*. Note that some security features (e.g., *parameterized prepared statement*) are provided by standard libraries of programming languages, such as Java.

5.2.2 Relation to the taxonomy

While the security frameworks support all top-level features from the taxonomy, we observed some noticeable differences to the literature. The selected security frameworks offer only 64% of the security features from our taxonomy. While nearly all features of *cryptography* are provided. The frameworks mainly lack sub-features concerning the three of the five top-level features, *authorization*, *secure data handling* and *system state protection* as visible in Figs. 6, 10 and 11.

It seems that many access control features might not be used in practice or did not reach practice, yet. The literature considers 11 access control features for *authorization* (see Fig. 6), but the security frameworks only offer 2 of these. In some cases, the selected frameworks might be able to realize security features such as *discretionary access control* or *rule-based access control* by utilizing other offered features. However, this was not mentioned in any of the documentation. Consequently, we did not label these security features in the taxonomy in Fig. 6 as being provided by the frameworks. In the case of *secure data handling*, 10 out of the 15 security features are provided by the frameworks, as shown in Fig. 10. For *system state protection*, the security frameworks offer 4 out of 8 security features we collected in the literature, as depicted in Fig. 11.

Additionally, our research suggests that security frameworks sometimes promote the application of various features as novel features, which may not align with our definition of security features according to literature. For example, Apache Shiro, Java EE, JBoss, and Seam Security offer a security feature called *remember-me* that they advertise as an *authentication* feature signifying an entity as "remembered from a successful authentication during a previous session" (The Apache Software Foundation 2010). Concerning our taxonomy in Fig. 8, this is not considered a security feature but implies a specific usage of *session management* features to keep a session open when an application is reopened. This suggests, that security frameworks have different views on what level of security features should be considered. Note that while some frameworks are not considered security frameworks, they may offer features that support the implementation of security features, or may even directly provide security features, even though these frameworks are not included in our list.

Finally, we found that many security features rely on manual implementation and are not offered by security frameworks. Most features for access control, some of secure data handling, and system state protection from our taxonomy have to be manually implemented without the use of a framework. Security monitoring features are offered by a few frameworks but might not be tailored to the needs of every project.

RQ2: We collected 44 security features from a set of 21 security frameworks identified in discussions on *Stack Overflow and Reddit*. The features overlap with the taxonomy obtained via the literature review. The most significant overlap occurs within the domains of *access control* and *cryptography*. Conversely, the least overlap is found in the domain of *security monitoring*. The relation of frameworks to the taxonomy is indicated in Figs. 6, 8, 9, 10 and 11 (features marked with **F**).

6 Manifestation of functional security features in source code (*RQ3*)

Since the goal of our work is to provide guidance in locating security features in source code, we need to understand how they manifest in codebases as this information can be utilized for creating traceability. To this end, we captured how each security feature in the analyzed security framework is provided to developers.

6.1 Security feature manifestation

As indicated in Table 6, we found that the security frameworks provide security features based on three mechanisms:

- *APIs* provide security features directly. A framework can define API classes, methods, or fields to invoke or configure security features. Listing 3 contains examples of the usage of APIs for encryption. As such, their usage is clearly visible within the codebase, which makes them easy to locate. This property can be leveraged to establish traceability, since APIs can serve as an entry point for the location of security features.
- *Configuration files* can be used to enable security features, potentially in addition to APIs, to provide configurations of used security features. Examples of such are given in Listings 4b and 7a. Developers can change values within a configuration file to modify specific values used by a framework. While they are clearly separated from the source code, they are still interacted with by the code base to fetch data that is relevant for security measures. Therefore, there is a need to connect the configuration file to the corresponding security features along with the API.
- *Annotations* are used by many programming languages, such as Java to extend functionalities of methods or classes within the implementation. A developer simply prepends a keyword with a corresponding marker, such as @, to the program element. Listing 2 shows examples of such annotations in lines 7 and 13. Security annotations can be used to either clearly define a context in which a method or class should be used, e.g., a security level (Peldszus et al. 2024), or to provide additional functionalities to methods or classes. Like APIs, they are clearly visible within the source code and can be used for tracing security features to locate their implementation.

Table 6 Security features provided by security frameworks via APIs (a), configuration files (c), and annotations (n)

	01 Spring Security	02 Apple Security Framework	03 Apache Shiro	04 JAAS	05 Java EE / Jakarta EE	06 Java Security Manager	07 OpenSSL	08 Windows .NET Identity Foundation	09 ASP.NET Membership Provider	10 ASP.NET Role Provider	11 OWASP ESAPI	12 JBoss Seam Security	13 Passport	14 Play Framework Secure Module	15 OACC	16 JGuard	17 Bouncy Castle	18 Endpoint Security Framework	19 EveryAuth	20 PicketBox	21 Sureness
access control																					
authentication	a/c	a	a/c	a/c	a/n				a		a/c	a	a	a	a	a/c		a	a	a/c/n	a
authorization	a/c/n	a	a/c/n	a/c	a	a		a/c		a	a	a/n	a	a/n	a	a/c	a		a	a/c/n	c/n
cryptography																					
cryptographic hashing	a	a	a		a		a				a/c	a				a/c	a				
encryption	a	a	a				a		a		a/c				a		a				
key management	a	a					a										a				
signature	a	a					a										a				
steganography																					
security monitoring																					
automated response						a															
history maintenance																					
logging	a/c/n						a				a/c	a/n						a	a	a	
system state protection																					
resource management																					
system state validation																					
session management	a/c	a	a/c		a/c/n						a/c	a	a	a	a				a		
state synchronization																					
secure data handling																					
data validation											a/c										
data sanitization											a										
retention control																					
secure storage		a													a						
trusted sources		a	a				a				a						a				

For each top-level security feature, Figure 14 shows how often each mechanism is utilized by the security frameworks to provide it. A security framework can provide the same security feature using multiple mechanisms.

All frameworks provide APIs for each security feature they offer. While many core functionalities of security features are implemented by using their APIs, configuration files are additionally used by less than half of the frameworks for each feature. Configurable values

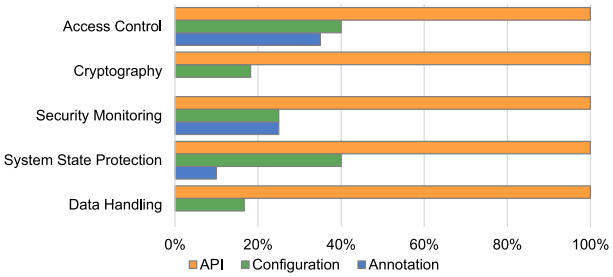

Fig. 14 Mechanisms used by security frameworks for providing security features for use in software systems

within these files are used by the security frameworks to allow simple modification of general parameters such as *timeouts* or used *encryption* algorithms. Finally, annotations are used for *access control*, *security monitoring* and *system state protection* features. Typically, classes or methods need to be annotated to extend the functionality of implemented *authentication* mechanisms or to enforce *authorization* rules.

6.1.1 Access control

All frameworks offer APIs for access control. As an example, user *credentials* can be generated in Apache Shiro by instantiating the class `UsernamePasswordToken` with a username and password. Then, the method `hasRole()` can be used to perform a role check.

40% of access control features include configuration files, which are often used to specify properties of authentication and authorization mechanisms. Picketbox uses configuration files to select login modules provided by the framework or to define roles that are then specified using Picketbox's API. In the configuration files provided by JGuard, the developer can select authentication schemes and define scopes. Spring Security also provides several configuration options through configuration files, such as the definition of a role hierarchy, which allows the inheritance of permissions between roles as illustrated in Listing 6.

```
<bean id="roleHierarchy" class="org.springframework.security.access.
    ↪ hierarchicalroles.RoleHierarchyImpl">
 <property name="hierarchy">
  <value>
   ROLE_DOCTOR        > ROLE_MEDICSTAFF
   ROLE_NURSE         > ROLE_MEDICSTAFF
   ROLE_MEDICSTAFF    > ROLE_STAFF
   ROLE_HOSPITALSTAFF > ROLE_STAFF
   ROLE_STAFF         > ROLE_UNAUTHENTICATED
  </value>
 </property>
</bean>
```

Listing 6: XML-configuration file showing the role hierarchy of our exemplary EHRS

In addition, annotations are typically used to restrict access to methods to a specific group of users. 7 of the selected security frameworks (Spring Security, Apache Shiro, Java EE, Secure Module, JBoss Seam Security, PicketBox, Sureness) make use of Java annotations to handle access control on the method level. For example, Spring Security allows developers to annotate methods with the annotation `@PreAuthorize` to restrict method invocations to a specified role given as a parameter, as we have shown in Listing 2 for different types of roles.

We also found that some features build up on a combination of an API and a configuration file. For example, OWASP ESAPI provides an API to handle login requests as a part of authentication. In a configuration file, the developer must set additional properties, such as maximum login attempts or timeout duration. In Apache Shiro, access control can be handled in multiple ways. For once, as explained before, an API and annotations can be used to perform role checks. Additionally, Apache Shiro offers a configuration file in which resources and authorization requirements, such as roles, can be defined.

Additionally, our example of the EHRS uses a combination of an API and a configuration file to realize access control in Spring Security. First, a role hierarchy is defined in a con-

figuration file, as shown in Listing 6. Then, methods are annotated using Spring Security's defined annotations such as in line 7 and 13 of Listing 2, thereby, referring to these roles.

In summary, APIs are mainly used to implement the main structure for authentication and authorization mechanisms. Annotations can be used to limit access for methods within the implementation and to decorate authentication mechanisms. Configuration files are used to provide additional configuration parameters, such as authentication schemes or properties.

6.1.2 Cryptography

Cryptography features are mainly realized by the usage of APIs. Several cryptographic methods are offered in Apple's Security Framework (iOS and OSX). The developer can, for example, use the method `SecKeyCreateEncryptedData()` to encrypt blocks of data using a public key and a given algorithm. In Listing 3a, we demonstrated how the EHRS from our example uses an API provided by Spring Security for encrypting data. There, we identified that even though it provides classes and methods for encrypting data, developers still need to write an implementation for some parts of the process, such as the generation of a salt in line 4. As the comparison between Spring Security and Bouncy Castle in Listing 3 shows, the extent of needed code can vary among different frameworks. APIs are used to sign and verify digital signatures in OpenSSL as well. A signature can be created by using the method `EVP_DigestSign` and verified using the method `EVP_DigestVerify`. Several key management features, such as the generation of random keys of a fixed length, are also realized through API usages in Spring Security. The class `BytesKeyGenerator` provides several methods for key generation, such as `generateKey()` or `secureRandom()`.

OWASP ESAPI and JGuard are the only security frameworks that provide configuration files for *encryption* and *cryptographic hashing* features. In the OWASP ESAPIs configuration file, it is possible to choose a cryptography algorithm used to encrypt or hash data. For example, a default hashing algorithm for passwords can be defined within a configuration file, while the method `cryptPassword()` can be used to hash passwords using the specified algorithm. In the same manner, the method `encrypt()` is used to encrypt plain text using the algorithm specified in the configuration file.

In general, cryptography features are provided via APIs to allow the encryption and signing of data. Configuration files can be used to set default algorithms for encryption, hashing, and more, as an alternative to setting them each time using method parameters. In total, 18% of identified cryptographic features can include configuration files, but these are only used for configuring low-level details of the feature usages.

6.1.3 Security monitoring

Logging is the security monitoring feature that is realized the most by the security frameworks. Logging is typically handled through the usage of an API. As an example for logging, EveryAuth offers a Boolean variable called `debug` to turn on and off logging. Similarly, PicketBox uses the class `PicketBoxLogger` to log certain predefined events. To define an *automated response*, the Java Security Manager throws a `SecurityException` once a security violation has been detected, which can then be used to specify response actions.

In OWASP ESAPI and Spring Security configuration files can be used for configuring a security event logger that is used over an API. Configuration parameters comprise secure

encoding of logged HTML messages and the definition of an upper bound of log file size. For example, in Spring Security, a configuration parameter must be set to enable the logging of authentication attempts.

Spring Security and JBoss Seam Security are the only security framework employing an annotation for security monitoring purposes. For Seam security, the documentation describes that the `@Logger` annotation can be used to inject a shared instance of a logger, avoiding configuring the logger in every class. This logger could also potentially be a secure logger that has been configured by the developer or follows a security pattern (Dougherty et al. 2009). However, no further information is given in the documentation.

To conclude, APIs are used to actively log events and define responses to security violations. Additionally, configurations are used to specify properties to modify and customize logs for specific purposes. Annotations only play a minor role in a few frameworks for security monitoring.

6.1.4 System state protection

Only *session management* features are offered by the selected security frameworks for the system state protection category. API calls are usually used to actively handle functionalities for persisting sessions, such as through the use of cookies. OWASP ESAPI, for example, offers the method `getCookie()` to receive a session cookie.

While the general functionality of session management features is realized through the use of APIs, 40% of the selected security frameworks offer configuration files to configure security features. As in the case of Apache Shiro and OWASP ESAPI, properties of session management features such as the duration until a session timeout, are often specified in a configuration file, as we have shown in Listing 4b. The actual sessions are implemented in other web frameworks such as the Jakarta XML Web Services (JAX-WS) of Java EE. The security frameworks target to secure the sessions of such web frameworks.

Java EE is the only security framework using annotations for persisting sessions by using the `@RememberMe` annotation. After annotating an implemented authentication mechanism class, the login is remembered by the system to keep the session persistent.

In summary, while APIs provide ways to manage sessions actively, configuration files are used to configure specific properties. Annotations can be used in addition to authentication mechanisms to keep a session open between actions.

6.1.5 Secure data handling

Along with cryptography, secure data handling features mainly rely on APIs for their realization. However, OWASP ESAPI is the only framework making use of configuration files for secure data handling to define rules for validation using regular expressions.

Apple's Security Framework provides the *secure storage* of data that is shared among applications over its keychain service. As shown in Listing 5, the class `SecKeychain` is used to store passwords, cryptographic keys, certificates, and notes.

Additionally, APIs are used to generate *timestamps* and provide a *source of randomness*, which are used by other security features, e.g., for logging or cryptographic purposes. Apache Shiro offers the method `getStartTimestamp()` for receiving the starting time of an opened session and a class `SecureRandomNumberGenerator` to generate secure random numbers.

APIs can also be used to set rules for regular expressions for validation features. For example, OWASP ESAPI explicitly offers a `ValidationRule` interface for specifying rules for data from an untrusted source. A corresponding configuration file is used to register these rules with the security framework.

In summary, secure data handling features, such as secure storage, generating data from trusted sources, and data validation and sanitation are mainly provided via APIs. Configuration files are only used in OWASP ESAPI for specifying validation rules through regular expressions.

6.2 Locating security features

The mechanisms used for integrating security features into a system are essential for locating the features. We observed that a majority of the implementations of security features are via API classes, methods, or fields. However, we still found a large number of additional functionalities of security frameworks that moved parts of the implementation, e.g., session management parameters and authorization policies, into configuration files and annotations. To estimate how well the different security features can be located, we investigated how their mechanisms can be used to map them to specific security features.

6.2.1 Source code APIs

Every security feature that we identified within the security frameworks is provided through an API. Since APIs are explicitly used in the implementation, their usages can be located by searching the source code. As cryptographic features, such as encryption and hashing mostly use APIs, their usages are easy to locate in principle. However, as also observed in existing works (Tuma et al. 2022), in some cases, APIs use the same method call for the realization of different security features. For instance, Bouncy Castle provides engine classes to realize different ciphers based on the method `init()` with a mode parameter for switching between encryption and decryption. This process is illustrated in Listing 3b, line 15, in which `ENCRYPT_MODE` denotes that a message should be encrypted. Similarly, messages can be decrypted by using `DECRYPT_MODE` as a parameter. The same API call realizes different features, and it becomes difficult to distinguish between them when no additional information is provided. Furthermore, the usage of API methods can only serve as an entry point for feature location techniques, since the security-critical code for using them also needs to be identified. Listing 3 shows that the amount of relevant code can range from a few code statements to complex configuration code as for Bouncy Castle. In summary, to facilitate the location of security features such as access control, cryptography, etc. from security frameworks, traceability can be established through the security frameworks' API calls.

```
# ESAPI Encryption
#
# The ESAPI Encryptor provides basic cryptographic
# functions with a simplified API. [...]
Encryptor.MasterKey=tzfztf56ftv
Encryptor.MasterSalt=123456ztrewq
# Provides the default JCE provider that ESAPI will
# "prefer" for its symmetric encryption and hashing.
# [...] Default: Keeps the JCE provider set to
# whatever JVM sets it to.
Encryptor.PreferredJCEProvider=
# By default, ESAPI Java 1.4 uses "PBEWithMD5AndDES"
# and which is very weak.
Encryptor.EncryptionAlgorithm=AES
# For ESAPI Java 2.0 - New encrypt / decrypt methods
Encryptor.CipherTransformation=AES/CBC/PKCS5Padding
```

(a) OWASP ESAPI configuration file showing configuration properties for using the encryption algorithm

```
public EncryptedPatientDataObject encryptPatientData(PlainText data) {
    EncryptedPatientDataObject encrypted_data = esapi.encrypt(data);
}
```

(b) API call of OWASP ESAPI encryption feature. Note, that no concrete cipher is specified here, since it is provided through the configuration file in Listing 7a

Listing 7: Encryption of patient data using the OWASP ESAPI encryption feature.

6.2.2 Configuration files

As described above, configuration files are mainly used for access control and system state protection features within the security frameworks. In principle, configuration files can be related to security features based on knowledge of the security frameworks. Some values in configuration files are only used for configuring security features that are provided through explicit APIs, whose location we discussed above. In such cases, no further configuration file-specific tracing is required. The challenge lies in locating the custom-implemented source code locations that interact with parts of the configuration file since configuration files typically do not require explicit interaction by the developer to provide some kind of functionality. Often, configuration files affect features simply by including them. For example, while the security-related parts of the Java EE API allow realizing session management, some lower-level features, such as session timeouts, can be set centrally in a configuration file of a security framework.

Configuration files can also contain references to relevant places in the code. For example, configuration files can apply features on namespaces or locations within the implementation, which can be used for tracing. OWASP ESAPI includes a structured configuration file listing configurable properties of features, which can be related to the implementation. As it is structured in different categories, it is possible to directly relate some of the categories to a specific feature. For example, the category `ESAPI Encryption` can be trivially mapped to the security feature encryption, since it contains different parameters for encryption features. Difficulties arise for developers when mapping lower-level security features to the configuration, as substantial security knowledge is required. Furthermore, as in the case of

source code APIs, some configuration options can be related to different security features as well. As shown in Listing 7a, the option `Encryptor.CipherTransformation` in the OWASP ESAPI configuration file defines what encryption should be used by default and could potentially point towards both a stream cipher or a block cipher, as the concrete cipher is specified by implementing the interface `Encryptor`, which aggravates the relation to the concrete security feature. When used in different parts of the system, the cipher to be used is not provided as a parameter, and the function call does not reveal the concrete low-level security feature in this case, as shown in Listing 7b. Therefore, source code APIs need to be considered as well in correctly identifying the concrete low-level security feature.

A significant challenge in identifying security features using configuration files arises when they need to be parsed. While APIs and annotations can be identified by parsing the code and traversing the abstract syntax tree using one of the many available parsers for each programming language, configuration files often use custom file formats and follow an individual structure. Therefore, configuration files require additional information and reasoning to map specific parts of them to security features.

6.2.3 Annotations

As described in Section 6.1.1, annotations can be used to apply security features to classes or methods. Mainly authorization on the method level, such as in Spring Security, shown in line 7 and 13 of Listing 2, and parts of authentication features can be identified by extracting annotations from the implementation. Logging and session management are other security features that can be realized by providing annotations. Traceability between source code and features can be achieved by annotating code with identifiers for features (Martinson et al. 2021; Bergel et al. 2021; Entekhabi et al. 2019; Andam et al. 2017). However, annotating every security-relevant line or block of code can be an exhaustive and tedious task, which creates a lot of overhead during development. As annotations of frameworks have unique names and correspond to specific security features, they can be seen as information-wise equivalent to feature annotations of tracing frameworks. These security features could then be traced by locating every occurrence of their respective annotations. As there is also no further code for using the security features, no further steps are required to provide a link between them. In the example of Spring Security shown in Listing 2, extracting the `@PreAuthorize` annotation for a method can clearly reveal which method is accessible by which role, thus providing a simple mapping between access control and the annotation.

RQ3: When security frameworks are used to integrate security features into a system, the extracted knowledge of how the frameworks provide security features can be used to locate features for all 5 top-level security features we identified. This knowledge encapsulates annotations, code APIs, and configuration files, as well as which interfaces provide which security features, in particular, the methods belonging to the APIs that provide a framework's cryptographic security features. However, embedding security features into the system requires writing code that actually uses the provided security features. This code may involve not only just calling the API, but also changing data formats or configuring security features such as supported authentication schemes or key lengths. Therefore, feature location techniques must also locate source code that is required to use the security frameworks, e.g., through information flow analysis of the data handed over to an API.

7 Application example

To demonstrate the utility of our taxonomy, we consider our exemplary EHRS. As we discussed throughout this work, the system utilizes a wide range of security features to handle the access and protection of patient data in a hospital. Recall that in the scope of the system, there are multiple roles that have access to the system. Among them are patient, doctor, and hospital staff. Through this system, a designated doctor is able to view patient data for which they have been granted permission by the patient. Other doctors should not be able to view the full extent of the patient's data. Only hospital staff is allowed to read and write data related to billing.

Selecting security features When starting the development of the EHRS, developers have to select the security features required to secure the system. The taxonomy of functional security features supports the reasoning of required security features by providing an overview of the types of security features that could be implemented. For example, based on the division of permissions illustrated in Fig. 1 and the covered features shown in Fig. 6, a combination of role-based access control and attribute-based access control is a suitable choice.

Due to the criticality of the system, it must be developed in compliance with relevant standards (United States Congress 1996; European Parliament and Council of the European Union 2007). For the US, for example, such a system would have to be developed in compliance with standards such as the NIST SP 800-53. The concrete standards to be followed depend on the concrete system and the target market. For illustration, we discuss how the mapping between the NIST SP 800-53, which only implicitly considers functional security features, and the security features in the taxonomy supports developers in selecting concrete security features that are required to address the high-level aspects considered in the standard. For example, the segment of the standard on system backups shown in Fig. 2 is mapped to the cryptographic security features of the taxonomy (Fig. 8). By following this mapping, the developers discuss possible functional security features and end up with deciding to encrypt backups using a block cipher.

The taxonomy and the mapping to high-level security standards provide a basis to developers for systematic reasoning about and selection of appropriate security features for developing a secure software system.

Realization of security features. To avoid insecure implementation of security features, developers can follow the best practice of using frameworks that provide ready to use implementations of the planned security features. To get an overview of possible frameworks, they could follow the mapping between the selected security features from the taxonomy and the security frameworks that provide those features. For example, they could look up which frameworks provide authentication features by following the mapping between the taxonomy and the frameworks. Based on this, they can see that the security feature is covered by several security frameworks and could decide to use Spring Security to realize role-based access control using an API, configuration file, and annotations.

The permission system have to be implemented by a developer, who must make a lot of considerations regarding the correct distribution of permissions to the users. One mistake in assigning permissions to users can have a significant impact, allowing unauthorized users to gain access to sensitive data. Still, developers have to write code to integrate the security framework into the system, which can be prone to errors. For example, to restrict access of a

method to specific roles, they must annotate the method with the correct roles. Additionally, they may define a role hierarchy in a configuration file, which allows the inheritance of permissions between roles by granting a child role the same permissions as a parent role.

> The mapping between security features in the taxonomy and which of those are provided by popular security frameworks helps developers in selecting appropriate frameworks for realizing the planned security features.

Certification of a software system Many critical systems must be compliant with the Common Criteria,[2] a requirement that is likely applicable to the EHRS as well. Among others, the CC requires the implementation of access control policies and functions, and provides details on how those security features must be realized. However, the huge size of the standard makes it challenging to identify those aspects that are relevant for the developed system. To systematically identify those aspects, developers can take the list of security features from the taxonomy that have been selected above and systematically look up the relevant locations in the standard using the mapping between the two. This way, they can effectively review whether all security features are implemented compliant with the CC, continuously throughout the software development life cycle. External auditors can then use the list of the implemented security features to identify the software components relevant for the security audit of the system.

> Using the mapping between the taxonomy and low-level security standards, developers can systematically assess all implementation details needed for a certification of a software system.

Locating security features in case of an incident Assuming an incident occurred, in which a nurse was able to access billing data related to the treatment of a patient, developers have to quickly react and recover all locations of the security features. A developer, who might (not) be familiar with the system, is tasked with resolving the cause of the incident. To support the investigation, the developer could leverage the taxonomy, which describes security features in a hierarchical order, helping them to reason about security features that might play a role in the incident. Based on the assumption of an authorization issue as a root cause of the incident, the developer could leverage feature location techniques which function as a search mechanism for identifying security features. Without requiring knowledge of these techniques, the developer can query for security features related to authorization, allowing the technique to automatically retrieve relevant locations of the usages of the authorization feature. In combination with these usage locations, they could investigate the source code providing the functionality that the nurse was able to execute. The analysis would show that the method `getBillingData()` is annotated with `@PreAuthorize{ROLE_HOSPITAL-STAFF}`, which allows users with the role hospital staff to access billing information. Since the method appears to be correctly annotated, there may be an issue with the role hierarchy.

Consequently, they additionally need to investigate the corresponding xml file, which configures the role hierarchy. Here, it becomes evident that according to the role hierarchy, the role nurse inherits all permissions from the role hospital staff. According to the use-case diagram shown in Fig. 1, only hospital staff should have access to billing data, not nurses. Due to the inheritance relation in the configuration, the nurse gained the permissions to read

[2] https://www.commoncriteriaportal.org/products/index.cfm

and write billing data. Therefore, the developer would need to fix the error by introducing two roles, medical staff and administration staff, which splits the granted access permissions to measurements and billing accordingly, and correct the role hierarchy as shown in Listing 8, as well as the annotation. This change would lead to the correct role hierarchy as shown in Listing 6. The differences in the distributed permissions are shown in Table 7.

```
@@ -1,8 +1,10 @@
  <property name="hierarchy">
   <value>
-   ROLE_DOCTOR          > ROLE_HOSPITALSTAFF
-   ROLE_NURSE           > ROLE_HOSPITALSTAFF
+   ROLE_DOCTOR          > ROLE_MEDICSTAFF
+   ROLE_NURSE           > ROLE_MEDICSTAFF
+   ROLE_MEDICSTAFF      > ROLE_STAFF
+   ROLE_HOSPITALSTAFF   > ROLE_STAFF
    ROLE_STAFF          > ROLE_UNAUTHENTICATED
   </value>
  </property>
```

Listing 8: Diff of a change in the role hierarchy

As discussed in Sec. 6, security features manifest in a system via the usages of APIs and annoations in source code as well as configuration files, which can be automatically detected using feature location techniques to help developers locate relevant security features. These techniques allow developers to retrieve feature locations without requiring knowledge of their internal workings.

Reasoning about related security features After the role hierarchy has been changed to fix the vulnerability discussed above, it is unclear what other parts of the system are impacted by this change. Here, developers may also need to investigate other related security features. For instance, while attribute-based access control and role-based access control may now be correctly implemented, developers should still check what other security features related to storing or retrieving data, such as secure storage, might be affected. Another entry point for further investigation could be the parent feature Authorization, which may be implemented either through custom code or other frameworks, which should all be checked. Recovering this code requires feature location techniques which must take into account API code, configurations, and annotations that security frameworks use to realize these security features.

The taxonomy provides developers with an concise overview of functional security features, which helps in reasoning about security features related with each other, i.e., since multiple variants of access control are combined.

8 Discussion and implications

The results of our study suggest the following implications for practitioners and research directions.

Table 7 Role-based access control policy for our EHRS. r = read, w = write, (r) = read if attribute is accepted; Bold permissions are explicitly specified and italic ones inherited

		Role	Diag.	Health Measurem.	Planning	Billing
Role Inheritance	Wrong	Doctor	**w/(r)**	w^1/**(r)**	w/r^1	w/r^1
		Nurse	none	**w**	w/r^1	w/r^1
		Staff	none	none	**w/r**	**w/r**
	Correct	Doctor	**w/(r)**	w^1/**(r)**	w/r^1	none
		Nurse	none	w^1	w/r^1	none
		(Med. Staff)[2]	none	**w**	w/r^1	none
		(Adm. Staff)[2]	none	none	w/r^1	**w/r**
		Staff	none	none	**w/r**	none

[1] Inherited permissions, [2] Abstract roles

8.1 Practitioners

Practitioners can use our results to better understand security features, their coverage by security frameworks as well as their relation to security standards. Because the taxonomy provides an overview of security features, concise explanations, and references to more detailed literature, it is also a good starting point for developers new to IT security. Our derived taxonomy indicated that for each functional security feature, there are many different sub-features relevant to practitioners, which need to be selected appropriately for each software system. The taxonomy offers a selection of needed security features on a high abstraction level, which are linked to multiple security standards. This facilitates security feature selection when working with security standards. Thereafter, as the taxonomy shows for which security feature a security framework exists, it is possible to choose an appropriate security framework based on the selection. For almost all of these security features, libraries and frameworks should be used to minimize risks for security issues through custom implementations.

Functional security features provided by libraries and frameworks can also be used as an entry point when performing code reviews. Our results show that configuration files are used by many security frameworks and play an essential role when realizing certain functional security features. Therefore, they should be reviewed as well. Aside from security features based on annotations, the use of security frameworks and APIs still requires a substantial amount of security-critical code, which is prone to insecurity and requires careful scrutiny. By identifying relevant code locations and configuration files, either on the entire project or when corresponding locations are changed, reviewers can be immediately pointed toward those locations. Thereby, identified code locations can be automatically related to the security features that are realized there instead of only low-level code statements that must be put into context manually.

Some of the security features from the taxonomy, e.g., multi-factor authentication, must be realized by developers by combining other security features. Here, the academic literature describes multiple helpful implementation-level security features by which frameworks should be extended to provide more straightforward use and reduce the probability of insecure implementations.

8.2 Researchers

As our results reveal that multiple security features are not offered by security frameworks, our taxonomy implies several research directions. In our SLR, we captured five kinds of functional implementation-level security features. While most features are provided by secu-

rity frameworks, despite best practices that advise otherwise (Jakobsen and Orlandi 2016), developers are likely to implement some of them on their own. Based on what the security frameworks provide, we assume *logging* and *data validation* features to be most likely not used as provided by security frameworks but to be mainly based on custom code. Additionally, we found multiple security features, such as *retention control* and *resource management*, to not be offered by any of the security frameworks. As such, it is essential to consider the tendency toward custom implementation of such security features in research on security compliance checks.

The reasons on why some security features are not included in security frameworks demand further investigation in future work. As security frameworks, such as Bouncy Castle or OpenSSL, are usually tailored towards specific use cases, some security frameworks implement some subfeatures of our taxonomy, but not all of them. Researchers should therefore investigate, whether security framework developers are not aware of these security features, or if there are other reasons for their exclusion.

Further, while our investigation provides a structured overview of security features available within security frameworks, future research should delve deeper into understanding their actual implementation and usage in practice. Specifically, this could involve conducting developer studies to understand how practitioners implement and adapt these features in practice or mining public repositories, such as GitHub, to identify security features in codebases. Our taxonomy provides a foundation for such investigations, enabling researchers to analyze security features in actual implementations.

The use of established security frameworks not only lowers the risk of security issues in a software system by avoiding custom insecure implementations of security features but also provides easy-to-locate entry points for feature location techniques. To improve the location of security-critical code, additional annotations for labeling source code can help identify relevant code portions. Still, developers should not be overwhelmed by too many additional annotations. Instead, information relevant to feature location can be gathered from security frameworks, as discussed in Section 6.

In the context of security audits or security compliance checks, the location of the source code portions corresponding to security features is essential. Our findings showed that concrete implementation-level security features might be relatively simple to locate. However, when looking at the literature on design-time security requirements (Jürjens 2005; Peldszus 2022), we notice a divergence in abstraction between the security requirements, e.g., declaration of what is sensitive information, and the concrete security features identified that will be used for implementing such security requirements. Following security by design techniques, security features are usually planned very abstractly but must be implemented taking a number of aspects into account to ensure that they are used securely and cannot be bypassed. This gap in abstraction is comparable to the differences observed above between high-level and low-level security standards and is a significant obstacle to checking the implementation for compliance with its security design (Peldszus et al. 2019, 2024; Tuma et al. 2022). Since our taxonomy of functional security features resides in between those two abstractions and effectively maps between them, our findings can be used as a basis for novel security feature traceability methods.

9 Threats to validity

We now discuss threats to validity.

9.1 Internal validity

Internal validity might be threatened by author bias. For once, the keywords that were used for the systematic literature research were chosen by the authors. This might additionally have an impact on the selection of the security standards, which were chosen based on the expert knowledge of the authors. To minimize the bias, we employed several authors from different research areas, such as the software engineering, security, and human factors domains, who held frequent discussions. Through this process, the paper selection revealed a large sample of security features considered in the literature. Therefore, relating the selected standards to our taxonomy still revealed a strong overlap while also providing more general terms for some categories within our taxonomy, confirming the representativeness of the sample. The same bias could also threaten the validity of the mapping between the standards and our taxonomy. To ensure the validity of the mapping, five authors held discussions on matches and discrepancies. Discrepancies that lead to changes were resolved by the same authors as well. Due to the open nature of the underlying discussions and achieving agreement being the process and not its outcome, we did not provide inter-rater agreements. This may limit the estimations of the challenges involved and the repeatability of this process.

A bias in selecting the security frameworks might be introduced by the Stack Overflow and Reddit security framework selection. Security frameworks discussed on these platforms may not accurately represent those widely used in industry, introducing a potential bias in our selection. These discussions may highlight frameworks with greater usability challenges or ones associated with popular programming languages. Additionally, thread recency could skew the results, favoring frameworks with active recent discussions while potentially excluding those still relevant but less frequently discussed. To address these potential biases, we expanded our search to include multiple developer communities, using Reddit alongside Stack Overflow to confirm that selected frameworks are relevant across different communities. Further, we might have wrongly excluded security frameworks based on our interpretation of the discussion in the threads. Similarly, the investigation of the homepage, reference guide, and the API documentation of the security frameworks for the framework security feature extraction could bias the resulting security features, as some security features might not have been considered as security features in the analysis process. To minimize the threat of wrongly excluding a security framework or security feature, two authors independently participated in the framework selection process, extracted the security features from the security frameworks, and held frequent discussions on the inclusion and exclusion of extracted features of the security frameworks.

A final threat to the correctness and completeness might be imposed by the sources used for the security feature search. The security features that can be extracted from the homepage, reference guide, and the API documentation of the corresponding security frameworks may not reveal a complete set of security features of each framework, as it might offer security features that are not well documented. Therefore, there may be a few security features offered by the security frameworks that we did not consider in this work. Still, with the selected security frameworks and sources, we were able to provide a rich comparison of their features to our taxonomy. Additionally, we were able to thoroughly reason about the realization of security features within commonly discussed security frameworks.

9.2 External validity

Multiple external factors threaten the generalizability of our results. The systematic literature research and security framework search might not allow us to capture a representative sample of the literature and security frameworks. Still, the chosen general keywords provide an extensive collection of literature and security frameworks, emphasized by the strong overlap between the taxonomy and security features from the security frameworks, as well as the security standards.

Generalizability might also be threatened by the content of the selected papers we investigated. A large overlap between the investigated security standards and the literature in the validation of the taxonomy revealed that this concern is not significant to our work. We can, therefore, conclude that our taxonomy covers a wide range of security features to be considered when implementing software systems.

Another threat to the generalizability of our results is introduced by utilizing Stack Overflow and Reddit as sources for the security framework investigation. The selection criteria for security frameworks (being mentioned in two or more threads on Stack Overflow and Reddit) might not reveal all popular frameworks. Since developers on Stack Overflow and Reddit mainly discuss frameworks available to everyone, we could have missed frameworks that are closed for public usage. Nonetheless, the investigated security frameworks contained a large number of security features, which are included in our taxonomy.

10 Conclusion

In this paper, we present a taxonomy of functional implementation-level security features based on an SLR of the literature, their mapping to widely used security standards, and their relation to popular security frameworks. Following an empirical approach, we aim to improve the understanding of the requirements for light-weight security feature location support. Our taxonomy contains 68 security features with the top-level features *access control*, *cryptography*, *security monitoring*, *system state protection*, and *secure data handling*. To examine which security features are contained in security frameworks commonly discussed on Stack Overflow and Reddit, we investigated existing security frameworks and related the provided security features to our taxonomy. While most functional security features considered in the literature are provided by security frameworks, there are still many that need substantial implementation effort by developers.

Finally, as a first step towards light-weight security feature tracing approaches, we investigated how security frameworks provide security features to developers and discussed strategies for locating security features to reduce the manual location effort to a minimum. We found, that security features provided by security frameworks mostly manifest in forms such as API calls, which are easy to identify in the codebase. As such, traceability techniques are able to leverage this information to enable the quick location of security features.

The practical implications show how developers can use our taxonomy to choose security features required to adhere to popular security standards and select appropriate security frameworks. We focused on the literature and security frameworks as reliable sources, constituting a self-contained study, still, follow-up work should investigate more data sources. A logical next step is an empirical investigation of the security features presented in the taxonomy with practitioners to identify challenges and best practices in implementing them. Future work should examine how these security features are applied in real software systems, either

through developer studies or by mining public repositories such as GitHub. Our taxonomy serves as a basis for this investigation, allowing researchers to assess the practical usage of both framework-provided and custom-implemented security features. An affirmation of its quality and usability would further support the claim of the practical implications.

Finally, we call for action to improve the location of security features while lowering additional development-time effort. Our findings build a foundation for this objective by providing a deeper understanding of implementation-level security features and which indicators could be used as entry points for their location. We hope that other researchers complement our taxonomy. Based on that foundation, we aim to develop methods that can be used to establish traceability between security feature models and their implementation in code.

Acknowledgements Supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

Data Availability A replication package of all the data of our systematic reviews of the literature, security standards and security frameworks is publicly available at Zenodo (Replication Package 2025).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Abbas A, Saddik AE, Miri A (2005) A State Of The Art Security Taxonomy Of Internet Security: Threats And Countermeasures. Computer Science

Abukwaik H, Burger A, Andam BK, et al (2018) Semi-Automated Feature Traceability With Embedded Annotations. In: International Conference on Software Maintenance and Evolution (ICSME), pp 529–533, https://doi.org/10.1109/ICSME.2018.00049

Acar Y, Backes M, Fahl S, et al (2017) Comparing The Usability Of Cryptographic APIs. In: 2017 IEEE Symposium on Security and Privacy (S&P), pp 154–17https://doi.org/10.1109/SP.2017.52

Adat V, Gupta BB (2018) Security In Internet Of Things: Issues, Challenges, Taxonomy. And Architecture. Telecommunication Systems 67(3):423–44. https://doi.org/10.1007/s11235-017-0345-9

Ahmadian AS, Peldszus S, Ramadan Q, et al (2017) Model-Based Privacy And Security Analysis With CARiSMA. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp 989–993,https://doi.org/10.1145/3106237.3122823

Andam B, Burger A, Berger T, et al (2017) FLOrIDA: Feature LOcatIon DAshboard for extracting and visualizing feature traces. In: International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM, pp 100–10https://doi.org/10.1145/3023956.3023967

Ardagna CA, Cremonini M, De Capitani di Vimercati S, et al (2009) Access Control in Location-Based Services, Springer, pp 106–126.https://doi.org/10.1007/978-3-642-03511-1_5

Baitha AK, Vinod S (2018) Session Hijacking And Prevention Technique. International Journal of Engineering & Technology 7(2.6):193–198

Batory D, Sarvela JN, Rauschmayer A (2004) Scaling Step-Wise Refinement. IEEE Trans Software Eng 30(6):355–371

Bau J, Wang F, Bursztein E et al (2012) Vulnerability Factors In New Web Applications: Audit Tools. Developer Selection & Languages, Stanford, Tech Rep

BBC (2020) Police Launch Homicide Inquiry After German Hospital Hack. https://www.bbc.com/news/technology-54204356/, [Online; accessed 04-December-2024]

ben Othmane L, Chehrazi G, Bodden E, et al (2015) Factors Impacting The Effort Required To Fix Security Vulnerabilities. In: Information Security, Lecture Notes in Computer Science, vol 9290. Springer, p 102–1https://doi.org/10.1007/978-3-319-23318-5_6

ben Othmane L, Chehrazi G, Bodden E et al (2017) Time For Addressing Software Security Issues: Prediction Models And Impacting Factors. Data Science and Engineering 2(2):107–124. https://doi.org/10.1007/s41019-016-0019-8

Bergel A, Ghzouli R, Berger T, et al (2021) FeatureVista: interactive feature visualization. In: ACM International Systems and Software Product Line Conference - Volume A. ACM, pp 196–20https://doi.org/10.1145/3461001.3471154

Berger T, Lettner D, Rubin J, et al (2015) What Is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In: Systems and Software Product Line Conference

Bertino E, Ghinita G, Kamra A (2011) Access Control for Databases: Concepts and Systems. Now Foundations and Trends

Bhanot R, Hans R (2015) A Review and Comparative Analysis of Various Encryption Algorithms. International Journal of Security and Its Applications 9(4):289–306

Bhatia T, Verma AK (2017) Data Security in Mobile Cloud Computing Paradigm: A Survey, Taxonomy and Open Research Issues. Journal of Supercomputing 73(6):2558–263. https://doi.org/10.1007/s11227-016-1945-y

Biggerstaff TJ, Mitbander BG, Webster DE (1994) Program Understanding and the Concept Assignment Problem. Commun ACM 37(5):72–82

Blythe JM, Sombatruang N, Johnson SD (2019) What Security Features and Crime Prevention Advice Is Communicated in Consumer Iot Device Manuals and Support Pages? Journal of Cybersecurity 5(1https://doi.org/10.1093/cybsec/tyz005

Bokhari MU, Shallal QM (2016) A Review on Symmetric Key Encryption Techniques in Cryptography. International journal of computer applications 147(10)

Bosch J (2000) Design & Use of Software Architectures-Adopting and Evolving a Product Line Approach. Pearson Education Ltd

Busch M, Wirsing M (2015) An Ontology for Secure Web Applications. International Journal of Software and Informatics 9(2):233–258

Chen K, Zhang W, Zhao H, et al (2005) An Approach to Constructing Feature Models Based on Requirements Clustering. In: International Conference on Requirements Engineering (RE), pp 31–40

Chung, Ferraiolo D, Kuhn D, et al (2019) Guide to Attribute Based Access Control (ABAC) Definition and Considerations.https://doi.org/10.6028/NIST.SP.800-162

Cornell D (2012) Remediation Statistics: What Does Fixing Application Vulnerabilities Cost. In: RSA Conference

Denker G, Kagal L, Finin TW, et al (2003) Security for DAML web services: Annotation and matchmaking. In: International Semantic Web Conference, Lecture Notes in Computer Science, vol 2870. Springer, pp 335–350

Denning DE (1976) A Lattice Model of Secure Information Flow. Commun ACM 19(5):236–24. https://doi.org/10.1145/360051.360056

Dent AW (2004) Hybrid cryptography. Cryptology ePrint Archive, Paper 2004/210, https://eprint.iacr.org/2004/210

Dit B, Revelle M, Gethers M et al (2013) Feature Location in Source Code: A Taxonomy and Survey. J Softw Maint Evol Res Pract 25:53–95. https://doi.org/10.1002/smr.567

Dougherty C, Sayre K, Seacord R, et al (2009) Secure Design Patterns. Tech. Rep. CMU/SEI-2009-TR-010, Carnegie Mellon University, Software Engineering Institute's Digital Librarhttps://doi.org/10.1184/R1/6583640.v1

Egele M, Brumley D, Fratantonio Y, et al (2013) An Empirical Study of Cryptographic Misuse in Android Applications. In: ACM SIGSAC conference on Computer & communications security, ACM, pp 73–84

Entekhabi S, Solback A, Steghöfer JP, et al (2019) Visualization of Feature Locations With the Tool FeatureDashboard. In: International Systems and Software Product Line Conference volume B. ACM, pp 1–4,https://doi.org/10.1145/3307630.3342392

European Parliament and Council of the European Union (2017) Regulation (EU) 2017/745 of the European Parliament and of the Council of 5 April 2017 on medical devices, amending Directive 2001/83/EC, Regulation (EC) No 178/2002 and Regulation (EC) No 1223/2009 and repealing Council Directives 90/385/EEC and 93/42/EEC. URL https://eur-lex.europa.eu/eli/reg/2017/745/oj, [Online; accessed 19-December-2024]

Fahl S, Harbach M, Perl H, et al (2013) Rethinking SSL development in an appified world. In: ACM SIGSAC conference on Computer & communications security, ACM, pp 49–60

Fang W, Miller BP, Kupsch JA (2012) Automated Tracing and Visualization of Software Security Structure and Properties. In: 9th International Symposium on Visualization for Cyber Security (VizSec). ACM, pp 9–16,https://doi.org/10.1145/2379690.2379692

Ferraiolo DF, Kuhn DR (2009) Role-Based Access Controls. ArXiv abs/0903.2171

Ghafir I, Prenosil V, Svoboda J, et al (2016) A Survey on Network Security Monitoring Systems. In: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), pp 77–82,https://doi.org/10.1109/W-FiCloud.2016.30

Glaser B (1978) Theoretical Sensitivity: Advances in the Methodology of Grounded Theory. Advances in the methodology of grounded theory, Sociology Press

Habiba U, Masood R, Shibli MA et al (2014) Cloud identity management security issues & solutions: a taxonomy. Complex Adaptive Systems Modeling 2:5. https://doi.org/10.1186/s40294-014-0005-9

Hakeem A, Shah M (2004) Ontology and taxonomy collaborated framework for meeting classification. In: International Conference on Pattern Recognition, IEEE, pp 219–222

Harbi Y, Aliouat Z, Harous S et al (2019) A Review of Security in Internet of Things. Wireless Pers Commun 108(1):325–344. https://doi.org/10.1007/s11277-019-06405-y

Harzing A (2007) Publish or perish. URL https://harzing.com/resources/publish-or-perish, online; accessed 20-December-2023

Hendre A, Joshi KP (2015) A Semantic Approach to Cloud Security and Compliance. In: Pu C, Mohindra A (eds) 8th IEEE International Conference on Cloud Computing (CLOUD). IEEE, pp 1081–108https://doi.org/10.1109/CLOUD.2015.157

Hermann K, Peldszus S, Steghöfer JP, et al (2025) An Exploratory Study on the Engineering of Security Features. In: International Conference on Software Engineering (ICSE)

Herzog A, Shahmehri N, Duma C (2007) An Ontology of Information Security. Int J Inf Secur Priv 1(4):1–23. https://doi.org/10.4018/jisp.2007100101

Hewett R, Kijsanayothin P (2009) On modeling software defect repair time. Empir Softw Eng 14:165–18. https://doi.org/10.1007/s10664-008-9064-x

Houmb S, Islam S, Knauss E et al (2010) Eliciting security requirements and tracing them to design: An integration of Common Criteria, heuristics, and UMLsec. Requirements Eng 15:63–9. https://doi.org/10.1007/s00766-009-0093-9

IBM (2023a) Discretionary access control (MAC). URL https://www.ibm.com/docs/en/zos/3.1.0?topic=controls-discretionary-access-control-dac, accessed: 2023-Dec-20

IBM (2023b) IBM Engineering Requirements Management DOORS Family. URL https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors/9.7.2, online; accessed 20-December-2023

IBM (2023c) Mandatory access control (MAC). URL https://www.ibm.com/docs/en/zos/3.1.0?topic=environment-mandatory-access-control-mac, accessed: 2023-Dec-20

Islam S, Mouratidis H, Jürjens J (2011) A framework to support alignment of secure software engineering with legal regulations. Software and System Modeling 10:369–39. https://doi.org/10.1007/s10270-010-0154-z

ISO/IEC JTC 1/SC 27 (2009) Common Criteria for Information Technology Security Evaluation. International Standard ISO/IEC 15408, International Organization for Standardization (ISO)

ISO/TC 22/SC 32 (2021) Road vehicles – Cybersecurity engineering. International Standard ISO/SAE 21434, International Organization for Standardization (ISO)

Jakobsen J, Orlandi C (2016) On the CCA (in)Security of MTProto. In: Workshop on Security and Privacy in Smartphones and Mobile Devices, p 113-116,https://doi.org/10.1145/2994459.2994468

Ji W, Berger T, Antkiewicz M, et al (2015) Maintaining Feature Traceability with Embedded Annotations. In: International Conference on Software Product Line. ACM, pp 61–7https://doi.org/10.1145/2791060.2791107

Jiao L, Hao Y, Feng D (2020) Stream cipher designs: a review. SCIENCE CHINA Inf Sci 63:1–25

Jin X, Sandhu R, Krishnan R (2012) RABAC: Role-Centric Attribute-Based Access Control. In: International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS),https://doi.org/10.1007/978-3-642-33704-8_8

Johns M, Braun B, Schrank M, et al (2011) Reliable Protection against Session Fixation Attacks. In: ACM Symposium on Applied Computing. ACM, SAC '11, p 1531-153https://doi.org/10.1145/1982185.1982511

Jürjens J (2005) Secure Systems Development with UML. Springer

Kamra A, Bertino E (2010) Privilege States Based Access Control for Fine-Grained Intrusion Response. In: International Conference on Recent Advances in Intrusion Detection. Springer-Verlag, Berlin, Heidelberg, RAID'10, p 402-421

Kang K, Cohen S, Hess J, et al (1990) Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

Kang W, Liang Y (2013) A security ontology with MDA for software development. In: International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC). IEEE, pp 67–7https://doi.org/10.1109/CyberC.2013.20

Katz J (2010) Digital signatures, vol 1. Springer

Kaur R, Singh A, Singh S, et al (2016) Security of software defined networks: Taxonomic modeling, key components and open research area. In: 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), pp 2832–283https://doi.org/10.1109/ICEEOT.2016.7755214
Khanam S, Ahmedy IB, Idris MYIB et al (2020) A Survey of Security Challenges, Attacks Taxonomy and Advanced Countermeasures in the Internet of Things. IEEE Access 8:219709–21974. https://doi.org/10.1109/ACCESS.2020.3037359
Kim A, Luo J, Kang MH (2007) Security Ontology to Facilitate Web Service Description and Discovery. Journal on Data Semantics 9:167–195. https://doi.org/10.1007/978-3-540-74987-5_6
Kour J, Verma D (2014) Steganography techniques–A review paper. International Journal of Emerging Research in Management &Technology ISSN pp 2278–9359
Krombholz K, Mayer W, Schmiedecker M, et al (2017) "I Have No Idea What I'm Doing" - On the Usability of Deploying HTTPS. In: 26th USENIX Security Symposium. USENIX Association, pp 1339–1356
Krueger J, Berger T, Leich T (2019) Software Engineering for Variability Intensive Systems, Auerbach Publications, pp 153–172.https://doi.org/10.1201/9780429022067-7
Kumar R, Goyal R (2019) On cloud security requirements, threats, vulnerabilities and countermeasures: A survey. Computer Science Reviews 33:1–4. https://doi.org/10.1016/j.cosrev.2019.05.002
Lazar D, Chen H, Wang X, et al (2014) Why Does Cryptographic Software Fail?: A Case Study And Open Problems. In: Asia-Pacific Workshop on Systems, ACM, p 7
Löhr H, Sadeghi AR, Winandy M (2010) Patterns for Secure Boot and Secure Storage in Computer Systems. In: 2010 International Conference on Availability, Reliability and Security (ARES), pp 569–573,https://doi.org/10.1109/ARES.2010.110
Mahapatra S, Singh B, Kumar V (2020) A survey on secure transmission in internet of things: Taxonomy, recent techniques, research requirements, and challenges. Arab Journal for Science and Engineering p 6211-6240
Martinson J, Jansson H, Mukelabai M, et al (2021) HAnS: IDE-based Editing Support for Embedded Feature Annotations. In: International Systems and Software Product Line Conference (SPLC), pp 28–31,https://doi.org/10.1145/3461002.3473072
McDonald N, Schoenebeck S, Forte A (2019) Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. Proceedings of the ACM on Human-Computer Interaction 3(CSCW).https://doi.org/10.1145/3359174
McGraw G (2004) Software security. IEEE Security & Privacy 2(2):80–83. https://doi.org/10.1109/MSECP.2004.1281254
Mirkovic J, Reiher P (2004) A Taxonomy of DDoS Attack and DDoS Defense Mechanisms. SIGCOMM Computer Communincation Reviews 34(2):39–53. https://doi.org/10.1145/997150.997156
Mukelabai M, Hermann K, Berger T et al (2023) FeatRacer: Locating Features Through Assisted Traceability. IEEE Trans Software Eng 49(12):5060–508. https://doi.org/10.1109/TSE.2023.3324719
Nadi S, Krüger S, Mezini M, et al (2016) Jumping Through Hoops: Why Do Java Developers Struggle With Cryptography APIs? In: International Conference on Software Engineering, ACM, pp 935–946
Oyetoyan TD, Cruzes DS, Jaatun MG (2016) An Empirical Study on the Relationship between Software Security Skills, Usage and Training Needs in Agile Settings. In: 2016 11th International Conference on Availability, Reliability and Security (ARES), pp 548–55https://doi.org/10.1109/ARES.2016.103
Oyetoyan TD, Jaatun MG, Cruzes DS (2019) Measuring Developers' Software Security Skills, Usage, and Training Needs. In: Exploring Security in Software Architecture and Design, pp 260–286,https://doi.org/10.4018/978-1-5225-6313-6.ch011
Patnaik N, Hallett J, Rashid A (2019) Usability smells: An analysis of Developers' struggle with crypto libraries. In: Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019). USENIX Association, pp 245–257
Peldszus S (2020) Development of Secure Software with GRaViTY. In: Workshop on Software-Reengineering & -Evolution
Peldszus S (2022) Security Compliance in Model-driven Development of Software Systems in Presence of Long-Term Evolution and Variants. Springer. https://doi.org/10.1007/978-3-658-37665-9_6
Peldszus S, Tuma K, Strüber D, et al (2019) Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings. In: International Conference on Model-driven Engineering Languages and Systems (MODELS). IEEE, pp 23–3https://doi.org/10.1109/MODELS.2019.00-18
Peldszus S, Bürger J, Kehrer T et al (2021) Ontology-Driven Evolution of Software Security. Data & Knowledge Engineering (DKE) 134:101907. https://doi.org/10.1016/j.datak.2021.101907
Peldszus S, Burger J, Jurjens J (2024) UMLsecRT: Reactive Security Monitoring of Java Applications with Round-Trip Engineering. IEEE Trans Software Eng 01:1–3. https://doi.org/10.1109/TSE.2023.3326366
Potter B, McGraw G (2004) Software security testing. IEEE Security & Privacy 2(5):81–8. https://doi.org/10.1109/MSP.2004.84

Ralph P, bin Ali N, Baltes S, et al (2021) Empirical Standards for Software Engineering Research. 2010.03525
Rana S, Parast FK, Kelly B et al (2023) A comprehensive survey of cryptography key management systems. Journal of Information Security and Applications 78:10360. https://doi.org/10.1016/j.jisa.2023.103607
Replication Package (2025) Replication Package. https://doi.org/10.5281/zenodo.11091429
Revelle M, Broadbent T, Coppit D (2005) Understanding Concerns in Software: Insights Gained from Two Case Studies. In: 13th International Workshop on Program Comprehension (IWPC). IEEE, pp 23–3https://doi.org/10.1109/WPC.2005.43
Riebisch M (2003) Towards a More Precise Definition of Feature Models. In: Modeling Variability for Object-Oriented Product Lines
Rivest RL (1990) CHAPTER 13 - Cryptography. In: van Leeuwen J (ed) Algorithms and Complexity. Handbook of Theoretical Computer Science, Elsevier, p 717–755,https://doi.org/10.1016/B978-0-444-88071-0.50018-7
Robillard MP, Murphy GC (2007) Representing concerns in source code. ACM Transactions on Software Engineering and Methodology 16(1):3. https://doi.org/10.1145/1189748.1189751
Robshaw M (1995) Block ciphers
Roth S, Gröber L, Backes M, et al (2021) 12 Angry Developers - A Qualitative Study on Developers' Struggles with CSP. In: Conference on Computer and Communications Security (CCS). ACM, p 3085-3103,https://doi.org/10.1145/3460120.3484780
Rubin J, Chechik M (2013) A Survey of Feature Location Techniques. In: Domain Engineering, Product Lines, Languages, and Conceptual Models
Russo ER, Di Sorbo A, Visaggio CA, et al (2019) Summarizing Vulnerabilities' Descriptions to Support Experts during Vulnerability Assessment Activities. Journal of Systems and Software 156(C):84-9https://doi.org/10.1016/j.jss.2019.06.001
Santos JC, Tarrit K, Mirakhorli M (2017) A Catalog of Security Architecture Weaknesses. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, pp 220–223
Santos JC, Tarrit K, Sejfia A et al (2019) An Empirical Study of Tactical Vulnerabilities. J Syst Softw 149:263–284. https://doi.org/10.1016/j.jss.2018.10.030
Schindler W (2009) Random Number Generators for Cryptographic Applications, Springer, pp 5–2https://doi.org/10.1007/978-0-387-71817-0_2
Schwarz T, Mahmood W, Berger T (2020) A Common Notation and Tool Support for Embedded Feature Annotations. In: ACM International Systems and Software Product Line Conference - Volume B. ACM, pp 5–8,https://doi.org/10.1145/3382026.3431253
Seiler M, Paech B (2017) Using Tags to Support Feature Management Across Issue Tracking Systems and Version Control Systems. In: Requirements Engineering: Foundation for Software Quality. Springer, pp 174–180
Shar LK, Tan HBK (2013) Defeating SQL Injection. Computer 46(3):69–77. https://doi.org/10.1109/MC.2012.283
Sparxsystems (2023) Enterprise Architect. URL https://www.sparxsystems.eu/, accessed: 2023-Dec-20
Stack Exchange I (2022) Stack Exchange API. URL https://api.stackexchange.com/, online; accessed 20-December-2023
Talooki VN, Bassoli R, Lucani DE et al (2015) Security concerns and countermeasures in network coding based communication systems: A survey. Comput Netw 83:422–44. https://doi.org/10.1016/j.comnet.2015.03.010
The Apache Software Foundation (2010) Apache Shiro - Simple. Java. Security. https://shiro.apache.org/, [Online; accessed 20-December-2023]
Tsipenyuk K, Chess B, McGraw G (2005) Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. IEEE Security & Privacy 3(6):81–84. https://doi.org/10.1109/MSP.2005.159
Tuma K, Peldszus S, Strüber D et al (2022) Checking Security Compliance between Models and Code. International Journal on Software and Systems Modeli. https://doi.org/10.1007/s10270-022-00991-5
United States Congress (1996) Health insurance portability and accountability act of 1996 (public law 104-191). URL https://www.govinfo.gov/content/pkg/PLAW-104publ191/pdf/PLAW-104publ191.pdf, [Online; accessed 19-December-2024]
Valente A, Holanda M, Mariano AM, et al (2022) Analysis of Academic Databases for Literature Review in the Computer Science Education Field. In: 2022 IEEE Frontiers in Education Conference (FIE), pp 1–7,https://doi.org/10.1109/FIE56618.2022.9962393
Venter HS, Eloff JHP (2003) A taxonomy for information security technologies. Computers % Security 22(4):299–30https://doi.org/10.1016/S0167-4048(03)00406-1
Vorobiev A, Bekmamedova N (2010) An Ontology-Driven Approach Applied to Information Security. Journal of Research and Practice in Information Technology 42

Xia X, Bao L, Lo D et al (2017) What do developers search for on the web? Empir Softw Eng 22(6):3149–3185. https://doi.org/10.1007/s10664-017-9514-4

Yassein MB, Aljawarneh S, Qawasmeh E, et al (2017) Comprehensive Study of Symmetric Key and Asymmetric Key Encryption Algorithms. In: 2017 International Conference on Engineering and Technology (ICET), pp 1–7,https://doi.org/10.1109/ICEngTechnol.2017.8308215

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Kevin Hermann[1] · Simon Schneider[2] · Catherine Tony[2] · Asli Yardim[1] · Sven Peldszus[1] · Thorsten Berger[1,3] · Riccardo Scandariato[2] · M. Angela Sasse[1] · Alena Naiakshina[1]

✉ Kevin Hermann
kevin.hermann@rub.de

Simon Schneider
simon.schneider@tuhh.de

Catherine Tony
catherine.tony@tuhh.de

Asli Yardim
asli.yardim@rub.de

Sven Peldszus
sven.peldszus@rub.de

Thorsten Berger
thorsten.berger@rub.de

Riccardo Scandariato
riccardo.scandariato@tuhh.de

M. Angela Sasse
martina.sasse@rub.de

Alena Naiakshina
alena.naiakshina@rub.de

1 Ruhr University Bochum, Bochum, Germany

2 Hamburg University of Technology, Hamburg, Germany

3 Chalmers University of Technology and the University of Gothenburg, Gothenburg, Sweden