FACULTY OF ENGINEERING
AND ARCHITECTURE

The Paved Path Methodology: A Human-Centered Approach to Software Security

Pieter De Cremer

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Computer Science Engineering

Supervisors
Prof. Bjorn De Sutter, PhD* - Matias Madou, PhD**

* Department of Electronics and Information Systems
Faculty of Engineering and Architecture, Ghent University

** Secure Code Warrior

December 2021

GHENT
UNIVERSITY

FACULTY OF ENGINEERING
AND ARCHITECTURE

The Paved Path Methodology: A Human-Centered Approach to Software Security

Pieter De Cremer

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Computer Science Engineering

Supervisors
Prof. Bjorn De Sutter, PhD* - Matias Madou, PhD**

* Department of Electronics and Information Systems
Faculty of Engineering and Architecture, Ghent University

** Secure Code Warrior

December 2021

GHENT
UNIVERSITY

ISBN 978-94-6355-555-5
NUR 958, 980
Wettelijk depot: D/2021/10.500/103

Always in motion the future is – Yoda

Acknowledgements

We acquire many new skills during life, and they are taught to us by many different people. Ever since I became a parent, I have realised just how much impact some of these people have had on me. Being a dad comes rather natural to me, and I can only attribute this to the many great examples I have had in my life of loving parents and grandparents. In the same way, I want to attribute the academic success of this work to the many great examples I have encountered during these last few years.

My thanks to...

Pieter Vandenbossche for standing on top of the desks, reading Latin texts in funny voices, and conducting an imaginary orchestra while playing Carmina Burana in class. My Latin teacher in high school taught me that interest and passion for a subject are infectious and to look for those teachers and students that can infect me.

Bjorn De Sutter for being such an infectious teacher. I found that paying attention to his classes took no effort at all and so I happily attended all possible elective courses taught by him. It is because of Bjorn that I found my way into the world of security.

Matias Madou for teaching me about hard work and determination, but also about not always taking myself too seriously. It was based on his advice that I applied for a grant and started this research. Matias is not only the world's okayest boss, but also a great mentor.

my examination board for their great feedback on both my research and this dissertation. I appreciate all the time and effort they spent evaluating this work.

the technical advisory board at SCW for their feedback on the intelligent tutor both in meetings and over mail. Brian's continued interest and feedback have been a source of confidence, and Chenxi's advice to *do the dumb thing first* will be a lesson that sticks with me for a long time.

the students, teaching staff, and security professionals who participated in the experiments and interviews in this dissertation.

my (former) colleagues at SCW and SysLab who made coming to work fun and relaxing. I love the atmosphere at the Bruges office, and the culture at SCW in general. Mondays would not be as fun without breakfast at work. And, it is always a pleasure to travel to Gent and catch up with my coworkers at the iGent tower. I already look forward to the next game night.

former classmates and housemates for helping me with group projects, helping me study, and for helping me unwind from time to time.

Brysen, Downey, and Tim for their feedback, insights, and assistance in my work, and for the countless evenings playing games to help me unwind.

Stéphanie for helping me improve my graphs, and for all her support and friendship.

Gillis for his feedback and support. For his friendship, and for always being there when I need a sympathetic ear. I wish everyone a friend as kind, down-to-earth, and awesome as Gillis.

my family for the unconditional support and love through all these years and for giving me a warm place to call home. When I am in need, there is never any doubt that my parents, my sister Laura, and her husband Bram are all ready to help however they can.

Sarah for being a firm believer, and a never ending source of support through the sometimes difficult times. Sarah and our daughter Suzanne are my biggest fans. I could not have finished this without them.

Op weg naar veilige software-ontwikkeling

Samenvatting

Het automatiseren van beveiligingstools heeft het mogelijk gemaakt om onveiligheden sneller en vroeger in de software ontwikkelingscyclus te detecteren. Desondanks zijn er nog steeds onveiligheden in bijna alle soorten software. De grote meerderheid van deze onveiligheden wordt veroorzaakt door fouten in de onderliggende code. Deze onveilige patronen in de code zijn al jaren gekend. Traditionele beveiligingstools kunnen deze problemen detecteren nadat de code is ontwikkeld, maar ze vertragen het ontwikkelingsproces en verhinderen het regelmatig lanceren van updates. Bovendien bieden ze geen specifieke hulp bij het oplossen van de gevonden onveiligheden. Wanneer de onveiligheden gedetecteerd zijn, is het aan de ontwikkelaars om deze op te lossen. Gemiddeld nemen bedrijven slechts één beveiligingsexpert aan per 75-200 ontwikkelaars. Het is eenvoudigweg niet mogelijk voor deze expert om elk van de ontwikkelaars hierbij te ondersteunen. Het is duidelijk dat softwarebeveiliging niet enkel nog de taak is van de expert. Het is onvoldoende om onveiligheden te detecteren, er moeten minder onveiligheden geschreven worden. Elke ontwikkelaar die code schrijft moet zelf verantwoordelijk zijn om dit vanaf het begin op een veilige manier te doen. Om hierop een impact te kunnen maken, moeten we kijken naar de betrokken processen, mensen, en technologie. Zo kunnen we garanderen dat er meer aandacht is voor softwarebeveiliging doorheen de hele software ontwikkelingscyclus.

Proces Ik stel een proces voor dat meer aandacht heeft voor de ontwikkelaar, genaamd de verharde-wegmethode. Met deze methode is het de bedoeling dat het beveiligingsteam niet langer de ontwikkelaars verplicht om beveiligingstools in te zetten. In de plaats daarvan moet een verharde weg gelegd worden voor de ontwikkelaars om te volgen. Deze verharde weg moet verschillend zijn voor elk project en hangt sterk af

van de gebruikte technologie. Ontwikkelaars en beveiligingsexperts moeten samenwerken om richtlijnen en patronen op te stellen die deze weg klaarleggen. Ze kunnen gezamenlijk beslissen over veiligheidskritische functionaliteit, bijvoorbeeld het beheer van encryptiesleutels. Ze doen dit door een bibliotheek en software te kiezen die hiervoor zal gebruikt worden. Ze kunnen zelfs een nieuwe bibliotheek ontwikkelen die eventueel een bestaande bibliotheek op een veilige manier aanroept. Ontwikkelaars zullen dan deze weg volgen, want deze bibliotheek is de eenvoudigste manier om nieuwe functionaliteit toe te voegen die beheer van encryptiesleutels vereist.

Mensen In de verharde weg methode zouden ontwikkelaars geen opleiding moeten volgen die eigenlijk bedoeld is voor beveiligingsexperts. Het doel van hun opleiding is niet om de veiligheid van de software te leren testen, maar hen de kennis en vaardigheden aan te leren die ze nodig hebben voor het ontwikkelen van veilige code. Daarom moeten ontwikkelaars een relevante en efficiënte opleiding ontvangen die specifiek aan hun rol is aangepast. Elke ontwikkelaar moet een defensieve opleiding volgen, in de taal en het raamwerk die gebruikt wordt tijdens hun dagelijks werk. Veel begrippen in softwarebeveiliging zijn algemeen toepasbaar, maar de oplossingen in de code zijn vaak specifiek gebonden aan de taal, en het zijn net die oplossingen die ontwikkelaars moeten aanleren.

Het opleidingsplatform van Secure Code Warrior (SCW) biedt zulke opleidingen aan in een brede waaier van programmeertalen. Daarbij voegen ze ook gamificatie toe om de ontwikkelaar te motiveren. Desondanks, is er een aanzienlijk deel van de gebruikers van het platform dat slechts een minimale hoeveelheid training volgt. De gebruikers volgen één van de vooropgestelde trajecten, en het is waarschijnlijk dat het tempo van deze opleidingen niet geschikt is voor iedereen. Sommige gebruikers vervelen zich door teveel herhaling, andere gebruikers raken gefrustreerd omdat de opleiding te snel moeilijk wordt.

Ik heb een intelligent leersysteem ontwikkeld voor het aanbevelen van oefeningen aan elke gebruiker op ieder ogenblik. Dit leersysteem gebruikt een Collaboratieve Filtering (CF) algoritme om aanbevelingen voor te stellen op basis van de voorkeuren van de meest gelijkgestemde gebruikers. Om dit algoritme aan te passen aan een leersysteem, worden gebruikers enkel als gelijkgestemd beschouwd wanneer zij hetzelfde nut ervaren van een oefening rond hetzelfde vaardigheidsniveau. Dit vaardigheidsniveau kan niet rechtstreeks gemeten worden, maar wordt regelmatig ingeschat door middel van het twee-parameter logistiek (2PL) model uit de Item Respons Theorie (IRT). Dit model beschrijft de relatie tussen de geobserveerde antwoorden van een gebruiker en diens vaardigheidsniveau. Door deze aanpassing aan leersystemen, kan de nauwkeurigheid van een CF algoritme verhogen tot meer dan 13%. Het definitief ontwerp van het intelligente leersysteem gebruikt het k-nearest neighbours baseline algoritme en bereikt een gemiddelde absolute afwijking van 0.4206 op beoordelingen op een schaal van 1 tot 5.

Technologie Meer traditionele beveiligingstools gebruiken een reactieve aanpak. Ze scannen (deels) afgewerkte code, en de oproepende context ervan, op zoek naar onveiligheden. De feedback van de tools komt vaak te laat, en dit vertraagt het regelmatig lanceren van updates. Het is geweten dat ontwikkelaars deze beveiligingstools storend vinden en zelfs vaak uitzetten. Ze worden beschouwd als één van de grootste belemmeringen voor de productiviteit.

In de verharde-wegmethode zijn tools in de eerste plaats ontworpen voor de ontwikkelaar. Daarvoor wordt gebruik gemaakt van een fundamenteel andere aanpak. In plaats van het zoeken naar onveiligheden, controleren ze het volgen van richtlijnen tijdens het schrijven van de code, ongeacht diens context. Wanneer ontwikkelaars bezig zijn met de functionaliteit van hun code, en hiervoor een bibliotheek gebruiken, dan staat de veiligheid vaak haaks op dit doel. Een goede tool zou de ontwikkelaar er op moeten wijzen wanneer die afdwaalt van de verharde weg, en die terugleiden zonder de productiviteit te schaden. Dit terugleiden van de ontwikkelaar op de verharde weg, zal diens productiviteit verhogen en tegelijkertijd de cognitieve belasting verlagen. Als de beveiligingsexpert de weg goed heeft aangelegd, dan zal de bekomen code veilig zijn.

In dit onderzoek heb ik geholpen bij het ontwerp en evaluatie van Sensei, een invoegtoepassing (Engels: plug-in) ontwikkeld door SCW, voor de applicatie die ontwikkelaars ondersteunt bij het schrijven van code. Net zoals een spellingscontrole programma, controleert Sensei of de code voldoet aan zogenaamde recepten. Het voorziet hulp bij het oplossen wanneer code afwijkt van deze recepten, in de vorm van kant-en-klare oplossingen (Engels: quick-fixes). Ik heb experimenten en gebruikerstests uitgevoerd die aantonen dat deze functionaliteit zeer bruikbaar is en snel aanvoelt als een verlenging van de bestaande ondersteuning voor ontwikkelaars.

Sensei voorziet een recept-verwerker die het mogelijk maakt voor ontwikkelaars en beveiligingsexperts om hun eigen project-specifieke richtlijnen in te stellen, in lijn met de verharde-wegmethode. Deze verwerker biedt suggesties aan uit de code, en toont een live voorbeeld van het effect van het recept op de code. In interviews met beveiligingsexperten geven zij aan dat Sensei de makkelijkste tool is die ze al gebruikt hebben voor het aanpassen van de opgelegde regels. In een empirisch experiment met studenten heb ik aangetoond dat deze aangepaste recepten een positief effect hebben op het gebruik van de tool met minimale impact op de productiviteit van de ontwikkelaar.

Toekomst Tot nu toe werden de opleiding en de tool apart beschouwd. In de praktijk is de grens tussen deze twee niet zo duidelijk. Ontwikkelaars leren vaak door te doen, en het leeraspect van de tools mag niet onderschat worden. Het intelligente leersysteem kan uitgebreid worden om data te gebruiken die verzameld wordt door Sensei en andere tools die gebruikt worden door ontwikkelaars zoals de code opslagplaats en de issue tracker. Ook kan in de toekomst de inschatting van het vaardigheidsniveau van de gebruiker uit het trainingsplatform gebruikt worden om de feedback van Sensei af te stellen.

Paving the path towards secure development

Summary

Automation of security tools has made it possible to identify software vulnerabilities faster and earlier in the Software Development Life Cycle (SDLC), but this has had little impact on the prevalence of vulnerabilities in almost all types of software. The vast majority (90%) of these vulnerabilities are caused by problems in the code, through insecure coding patterns that have been known for years. Traditional security tools are capable of detecting these problems after the code has been developed, but they slow down agility and release cycles. Additionally, they do not provide specific guidance to remediate the found vulnerabilities. Once the vulnerabilities are discovered, it is up to the development team to fix them. On average a company hires only 1 security expert for every 75-200 developers. This expert simply cannot assist each of those developers. It is evident that security is no longer just the responsibility of the expert. The ability to detect vulnerabilities is not enough; we need fewer vulnerabilities to be created. Every software developer producing code should be responsible for doing this securely from the start. To make impactful changes, we have to look at the processes, the people, and the technology involved, to guarantee better software security throughout the whole SDLC.

Process I propose a more developer-friendly workflow, named the paved path methodology. In this methodology, the security team should not force security testing on developers, but instead gradually build a paved path for developers to follow. This paved path should be different for each project and heavily depends on the technology stack for that project. Together, developers and security experts should build standards and patterns that lay out the paved path. They can decide together how security critical features, such as key management, should

be handled. They do this by deciding on the library and the tools needed, or even by creating a new (wrapper) library. Developers will then follow that path, as using this library is the easiest way for them to implement a feature that needs key management.

People In the paved path methodology, developers should not be handed repurposed education meant for security professionals. The goal of their education is not to teach them to *test* the security of the code, but to teach them the knowledge and skills necessary to *produce* secure code. The developers should hence be provided with role-specific, relevant, and efficient training. Each developer should receive defensive training in the same programming language and framework they are using daily in order to understand syntax specific secure and insecure coding patterns. While many security concepts are generally applicable, the actual solutions to problems are often programming language specific, and these solutions are exactly what developers should be taught.

The Secure Code Warrior (SCW) training platform provides such defensive training in a wide range of programming languages. Additionally, it provides some gamification features to keep the developers engaged. Despite that, there is still a significant part of the user base that only follows a minimal amount of training. Users follow one of the predetermined courses, and it is likely that the pacing of these courses does not fit their needs. Users get bored due to too much repetition, or frustrated because the content is moving too fast.

I created an Intelligent Tutoring System (ITS) to recommend exercises to each individual at any point in time. This ITS uses a Collaborative Filtering (CF) algorithm to make a recommendation based on the preferences of the most like-minded users. To adapt this algorithm to learning systems, users are only considered like-minded if they find an exercise similarly useful around the same ability level. This ability level cannot be observed directly, but is regularly estimated by using the two-parameter logistic (2PL) model from Item Response Theory. This model describes the relation between the observed answers of a user, and their ability level. By using this adaptation to learning systems, the performance of a CF algorithm can be significantly improved, by more than 13%. The final design of the ITS uses a k-nearest neighbours baseline algorithm and reaches a mean absolute error of 0.4206 on a rating scale from 1-5.

Technology Traditional security tools use a reactive approach, scanning (partly) completed code and its calling context for vulnerabilities. The feedback they provide comes too late, slowing down deploy and release cycles. Developers are known to dislike and often disable these security tools during development. They frequently perceive the tools as one of the biggest inhibitors of productivity.

In the paved path methodology, tools are in the first place designed as developer tools, using a fundamentally different approach. Instead of scanning for vulnerabilities, they enforce guidelines regardless of context as the code is being written. When developers are focused on the functionality of their code and using a library for this purpose, security is usually orthogonal to that purpose. A good tool should then warn a developer when they stray from the paved path and guide them back without hurting productivity. This guiding of the developer along the paved path, boosts their productivity while lowering their cognitive burden. If the security experts have done a good job laying out this paved path, the resulting code will be secure.

In this research, I helped design and I evaluated the Sensei Integrated Development Environment (IDE) plugin, developed by SCW. Sensei enforces so-called recipes in the IDE, similar to an as-you-type spellchecker. It also provides remediation guidance in the form of quick-fixes when these recipes are violated. I conducted experiments and usability tests that show that these features are usable and quickly feel like a natural extension of the existing toolkit of the developer.

Sensei provides a recipe-editor to allow developers and security experts to create their own project-specific guidelines, in line with the paved path methodology. The editor can generate suggestions from the context of the code and provides the recipe-writer with a live preview of the recipe, showing its markings on the code. In interviews conducted during this research, security professionals indicate that customizing recipes with Sensei is easier than any other tools they have used in the past. Furthermore, in an empirical experiment with students I have shown that customized recipes are effective at keeping the developer's trust with minimal impact on the developer's productivity.

Perspectives Until now, education and tools were considered two separate things. In reality, the border between these two is not that clearly defined and they blend over into each other. Developers often learn while doing, and the educational aspect of Sensei itself should not be underestimated. In the future, the ITS can be extended to use information

gathered by Sensei and other developer tools such as the code repository and the issue tracking system. At the same time, the ability estimate of the training platform can be used to tune the feedback of tools such as Sensei to the ability of the user.

Contents

List of Tables

List of Figures

Acronyms

1PL one-parameter logistic model. 47, 48, 52, 63

2PL two-parameter logistic model. 46–49, 52, 53, 57, 58, 61–63, 70, 91, 92, 97, 99, 100, 146, 191, 209–211

3PL three-parameter logistic model. 52, 53

4PL four-parameter logistic model. 52, 54

AI Artificial Intelligence. 189

ANOVA Analysis Of Variance. . . . . XIX, 64, 65

API Application Programming Interface. . 11, 15, 30, 112, 116, 118, 119, 123, 125, 130–133, 160, 165, 169, 170, 172–174, 183, 188, 189, 195–197, 202, 213, 241, 247, 250, 251

ASIDE Application Security plugin for Integrated Development Environment. 169

AST Abstract Syntax Tree. 121, 187, 199, 250

AWS Amazon Web Services. 118

BBC The British Broadcasting Corporation. 22

CAT Computerized Adaptive Test. 44, 45, 58, 71, 72, 94, 99

CF Collaborative Filtering. 33, 34, 36–38, 40, 41, 43, 45, 51, 54, 57, 61, 74–76, 80–84, 87, 91, 94, 96, 99, 209–211, 228, 231

CI Continuous Integration. 193, 244

CICD Continuous Integration and Continuous Delivery. 3, 4, 135, 162, 172, 247

CML Conditional Maximum Likelihood. 49, 50

CNCF Cloud Native Computing Foundation. 204, 205

CPU Central Processing Unit. 202

CTT Classical Test Theory. 35, 51, 52

CVE Common Vulnerabilities and Exposures. 92

CWE Common Weakness Enumeration. 30, 92

DES Data Encryption Standard. 195–198, 250

DoS Denial of Service. 43

EC European Commission. 185

EE Enterprise Edition. 30, 68, 73, 82, 118, 237

EFP Effective False Positive. . 125–127, 161, 163, 167, 169, 172, 173, 175, 191, 195, 201, 242, 244, 246

EM Expectation-Maximization. 50

EO Executive Order. 186

ESAPI Enterprise Security API. 112, 125, 130, 189

ESF European Social Fund. 15

ESVD Early Security Vulnerability Detector. 169

EU European Union. 185

FCP Fraction of Concordant Pairs. 84, 228

FIT Fixed Item Test. 44, 45

FOD Fortify on Demand. 194, 248

FSA Fortify Security Assistant. 194, 248

FSCA Fortify Static Code Analyzer. 247, 248

GDPR General Data Protection Regulation. 56, 185, 186

GUI Graphical User Interface. . . . XXII, 111–114, 126–128, 166, 198

HIPAA Health Insurance Portability and Accountability Act. . .186

HTML Hyper Text Markup Language. 123, 124, 170

HTTP Hyper Text Transfer Protocol. 129

IaC Infrastructure as Code. 183, 202

id identifier. 56, 67

IDE Integrated Development Environment. 12, 13, 99, 100, 104, 108, 109, 111, 113, 116, 118, 121–124, 132, 133, 136–138, 140, 141, 151, 156, 159, 161, 164, 167, 168, 171, 172, 175, 187, 189, 194, 198, 209, 212, 239–241, 243–246, 248–252

IEC International Electrotechnical Commission. 185

IRF Item Response Function. 46–49, 52–54, 58

IRT Item Response Theory.33, 35, 45, 46, 49–53, 55, 57, 58, 62, 64, 72, 73, 95, 98, 99, 209, 210

ISO International Organization for Standardization. 185

ITS Intelligent Tutoring System. . . 13, 17, 20, 33–36, 40, 42, 44–46, 52–54, 56, 57, 61–63, 72, 74, 91, 97, 100, 172, 209–211

JAR Java ARchive. 252

JDK Java Development Kit. 130, 189

JML Joint Maximum Likelihood. 49

JSF JavaServer Faces. 68, 69

JSP JavaServer Pages. 30, 137, 138

k-NN k-nearest neighbours. XIX, 40, 61, 76, 77, 79, 82, 85, 86, 88–91, 94, 95, 228, 229, 231

MAE Mean Absolute Error. 83–86, 90, 228

MML Marginal Maximum Likelihood. 49, 50

MSD Mean Squared Difference. 77, 85

MVC Model View Controller. 30

NIST National Institute of Standards and Technology. . . .186, 204

NMAE Normalized Mean Absolute Error. 83

NMF Non-negative Matrix Factorization. 81

NNMF Non-negative Matrix Factorization. 41

O&O Onderzoek & Ontwikkeling *(English: Research & Development)*. 175

OPA Open Policy Agent. 205, 207

OS Operating System. 125, 127, 132, 138, 141, 175, 201–203, 234, 237

OWASP Open Web Application Security Project. 26, 30, 31, 44, 62, 67, 85, 92, 93, 100, 112, 125, 130, 188, 189, 191, 193, 203–205, 207

PCI DSS Payment Card Industry Data Security Standard. 185

PMF Probabilistic Matrix Factorization. 41, 80, 81

QA Quality Assurance. 184

RMSE Root Mean Squared Error. 83, 84, 228

ROI Return on Investment. 156, 171, 177, 185

SaaS Software as a Service. 202, 249

SAST Static Application Security Testing. . .193, 241–249, 251, 252

SBOM Software Bill of Materials. 186, 191

SCA Software Component Analysis. 191, 240, 241

SCW Secure Code Warrior. . . . . XXII, 1, 13–15, 20–25, 27–31, 33, 42–44, 47, 49, 53, 56, 57, 61–64, 74, 76, 80, 85, 87, 91, 92, 94–97, 99–101, 104, 108, 111, 112, 118, 123, 124, 133, 136–138, 146, 151, 152, 156, 162, 171, 175, 204, 209–212, 216

SDK Software Development Kit. 118

SDLC Software Development Life Cycle. . 71, 105–107, 121, 160, 162, 167, 171, 172, 178, 184, 187, 192, 194, 195, 211, 239–252

SQL Structured Query Language. 17, 23, 27–29, 42, 43, 100, 137, 141, 144, 146, 164, 171, 196

SVD Singular Value Decomposition. 41, 80, 81

UI User Interface. 135, 151, 152, 154, 160, 166, 167, 174, 176, 199, 209, 212, 244

URL Uniform Resource Locator. .140

US United States. 186

UX User Experience. 151, 152

VG Veracode Greenlight. 249

VLAIO Vlaams Agentschap Innoveren & Ondernemen *(English: Flanders Innovation & Entrepreneurship)*. 174

VM Virtual Machine. 202, 203

VMI Virtual Machine Image. 202

VSA Veracode Static Analysis. .249

WLE Weighted Likelihood Estimator. 51

XML Extensible Markup Language. 17, 108, 121, 122, 137, 183, 198, 247, 252

XSS Cross-Site Scripting. 6, 67, 137, 146, 170, 175, 189, 211

XXE XML External Entity. 66, 92, 234

YAML YAML Ain't Markup Language. .111, 113–115, 126–129, 135, 165, 174, 183, 198–200, 209, 237, 238, 244, 250, 252

Glossary

AppSec Application security team. This term emphasizes the context of security regarding software applications, in contrast with network security. 2

bug tracking system A software application used to keep track of reported bugs in software development projects. Often this is not a standalone application but part of an issue tracking system. 2, 4, 107

Dev Development team. This team writes the code of the application. 2, 3

DevOps Contraction between Dev and Ops, a more or less unified team of developers and operators. 3, 4, 106, 112, 190

DevSecOps Contraction between Dev, Sec, and Ops, a more or less unified team of developers, security experts, and operators. 4, 5, 108

exploit A sequence of commands, inputs, or other manipulations that can take advantage of a security problem and cause harm to the stakeholders of the application. 6, 105

flaw A mistake in the design of the application that can lead to unwanted behaviour. They are mistakes in the functional structure of the application that are harder to detect automatically than security bugs. 66

issue A unit of work to accomplish an improvement in a software development system. An issue can be a bug, a requested feature, documentation, and more. 2

Ops Operations team. This team is responsible for running the code so that the application is available to customers. 3

Sec Security team, in the context of this book used to indicate the application security team. This team has access to the code and is responsible for evaluating and monitoring its security. . . 2, 4

security defect A security problem that has not yet been proven to lead to a vulnerability. 6, 157

security problem Overlapping term for security bugs and security flaws. 2, 4, 22, 106, 107

Slack A popular business communication platform that consists of persistent chat rooms organized by topic, private groups, and direct messaging.. . . . . 159, 166

vulnerability A security problem for which it has been proven an exploit exists. 1, 2, 6, 12, 23, 56, 105, 106

Chapter 1

The paved path methodology

You must unlearn what you have learned.

Yoda

Security automation has made it easier to identify software vulnerabilities, but this has had little impact on the prevalence of vulnerabilities in almost all types of software. To turn the tide, fundamental changes need to be made to software development practices. The ability to detect vulnerabilities alone is not enough, we need better processes and tools to prevent and fix vulnerabilities in a scalable way.

During the past four years I have researched developer and security practices while working for the company Secure Code Warrior (SCW)[1], in collaboration with Ghent University. During this time I built a vision of collaboration between developers and the security team, which I have named the paved path methodology. In this chapter, I describe the observations made during my research and the vision I have built. In the remainder of this work, I explain how this vision can be achieved through more intentional education (Part I) and tools (Part II).

[1] `https://www.securecodewarrior.com/`

If nothing else, take away from this chapter...

With the paved path methodology, I have built a vision to make software security a shared responsibility between the security team and developers. When using this methodology, the security team should not force security testing on developers, but instead gradually build a paved path for developers to follow. Developers will then follow that path, as it is the easiest way to achieve their goals. The discussed practices make it easier for developers to produce secure code and fix existing vulnerabilities in a scalable way without harming their productivity. To support the paved path methodology better education and tools should be provided that are more human-centered and keep the developer experience in mind.

1.1 A story of increasing collaboration

1.1.1 The security team

Security issues still exist in all software products: 100% of the applications tested by Trustwave in 2017 displayed at least one vulnerability [1]. 90% of these vulnerabilities are caused by problems or oversights in underlying code [2]. They are the results of mistakes made by the programmers during development. These are not new problems, the same types of vulnerabilities have been widely present in software for decades.

The (application) security team (Sec or AppSec) at these companies is responsible for evaluating the software and finding all the vulnerabilities. With the use of security tools much of this process is automated and so they have become quite competent at finding problems in the code. In fact, many of the reported numbers are collected through security tools used by these very teams [3].

The ability to detect security problems alone is not sufficient, more focus should be on preventing and fixing them in a scalable way. Once these (potential) vulnerabilities are discovered, it is up to the development team (Dev) to fix them. In order to help them manage this task, the security team pushes discovered vulnerabilities into a bug tracking system. They even organise the vulnerabilities by category and prioritize them by severity of impact. To actually understand each issue, and to fix them in a consistent way, however, developers are often on their own. On average a company hires only 1 security expert for every 75-200 developers [4–6]. This expert simply cannot assist each of those developers. It is evident that security is no longer just the task of this expert. Every developer should be responsible for producing code securely from the start.

1.1.2 The development team and the operations team

In order to make producing secure code more scalable, we can take a look at the improved collaboration between developers and operators in the DevOps movement. The operations team (Ops) used to be the only one responsible for testing and deploying code delivered by developers. Once the code was finished, and working on the developer's local machine, it was *thrown over the wall* for the operators to deal with. The DevOps movement aims to make this a more shared responsibility between Dev and Ops. Ideally, the two become one integrated DevOps team. In reality they are often still two closely related teams.

This close collaboration between the two is key. The operations team provides a service to developers that enables them to test and deploy their own code. They do not force automation but gradually build a Continuous Integration and Continuous Delivery (CICD) pipeline. This pipeline is different for each project and heavily depends on the chosen technology stack for that project. Through this pipeline, developers are able to check in smaller pieces of code more frequently and quickly receive feedback. DevOps has quickly gained popularity, 43% of developers report using its practices and 80% of developers think using DevOps practices is important [7].

The automated CICD pipeline also benefits the security team. Because of the faster release and deploy times, fixed security problems find their way into production faster. The CICD pipeline also allows them to automatically run static and dynamic analysis tools more easily. On bigger projects, such tools easily need a couple of hours to complete their analyses. This is not ideal for DevOps pipelines, where tight feedback loops are important. Almost all developers (96%) report that the biggest inhibitor to productivity is the disconnect between development and security workflows [6]. A guideline that is sometimes mentioned for DevOps tools is the coffee test. The idea behind this test is that all automations should be finished within the time it takes a developer to get a cup of coffee after checking in their code.

To solve this, the security team can tune the tools so that they run more lightweight analyses. One way to do this is by disabling certain rules. More lightweight analyses do not hinder the developers' productivity and also allow for fast feedback of the analysis results. Of course, the full scan should still be run regularly, for example on a daily basis. Faster feedback through lightweight analyses is definitely an improvement over delayed pushing of security problems into a bug tracking system, but it is not enough. The CICD pipeline may be convenient to automate security, but it is still disconnected from the development workflow. There is still only one expert to help up to 200 developers, so the problem of preventing and fixing security problems at scale remains. The security team acknowledges this, as they rank creating developer-friendly workflows as their top priority, even ahead of protecting the production software itself [6].

1.1.3 Three is a party

Similarly to the DevOps movement, security should become a shared responsibility between the development (or DevOps) team and the security team. The security team should not generate reports and *throw it over the wall* to developers. Instead, they should closely collaborate with developers to enable them to produce secure code consistently. This can be achieved through the *paved path methodology.*

Like the operations team, the security team should provide a service to developers to make it easy for them to secure their own code in a consistent way. They should not force security testing on developers, but instead gradually build a *paved path* for developers to follow. Just like the CICD pipeline, this paved path should be tailored for each project and will heavily depend on the technology stack for that project. Together, developers and security experts should build standards and patterns that lay out the paved path for their project. For example, they can decide together how key management should be handled by selecting the library and the tools needed. The correct way to handle key management will depend on the programming language, the framework, and the type of software application. Developers will then follow the paved path, as it is the easiest way for them to implement a feature that needs key management. Automated checks can be included in CICD by operators to prevent any other ways of handling key management.

In order to achieve this close collaboration, the security team needs to be deliberate in their approach. Their focus should not just be on the code, but also on the developer. They should be mindful of their communication and be empathetic. This way Sec and DevOps can truly come together to form a DevSecOps team with aligned goals.

1.2 Improved culture

This shift towards collaboration starts with a shift in culture. Historically, the development team and the security team have developed somewhat of an adversarial relationship. The security team has to constantly fight to make security a priority during development. They are aware of the risks and the costs related to poor security and they try their hardest to find all the problems in the code.

Since the security experts are understaffed and unable to adequately assist developers, all they end up doing is slapping developers on the wrist by pointing out their mistakes. In doing so, the security team loses an opportunity to build a trusted relationship and provide a valuable service to developers. This kind of interaction understandably causes resistance and even contempt from the developers towards the security team. This in turn gives security experts the impression that developers do not care about security. Furthermore, it is often the case that security experts do not have sufficiently intimate knowledge of software development. Even with the right intentions, they commonly lack the skills, or at the minimum the credibility, to properly advise developers in improving development processes and standards.

There is a clear need for mutual respect, empathy, and better cooperation. The security team needs to empathize with developers and provide a meaningful service to them. But to do so effectively, the developers need to empathize with the security team and work constructively to make security an inherent part of the software development process. This culture of cooperation and empathy is started by aligning goals, and aligning language among the two teams.

1.2.1 Aligning goals and metrics

To make security a shared responsibility, we have to meet developers in the middle. Developers, and the business as a whole, want to ship features regularly and with predictable speed. It is hurtful to the business to delay releases for security concerns when customers are promised these new features. The security team needs to understand this, and get involved from the start. Security becomes a shared responsibility, but so does building and deploying fast. Moreover, faster building and deploying also benefits security, as any security problems that eventually show up in production can also be patched and updated faster.

The mutual goal of a DevSecOps team should be to reduce the number of vulnerabilities in later stages of the development life cycle, while also still maintaining or even improving deployment metrics [8]. It then becomes clear that improving one at the detriment of the other, is not real improvement.

1.2.2 Aligning communication

The culture of mutual empathy is also improved with more deliberate and conscious communication. Instead of shaming or even punishing developers when problems are introduced in the code, the security team should try to understand the developer's challenges, and offer help. When the paved path has been built through collaboration, this implies the guidelines for the project have been mutually agreed upon. When such guidelines are violated, it becomes easier to demonstrate empathy and to show good intent. The security team can ask developers if the guidelines are insufficiently clear, if they lack recommendations for specific edge cases, or if there is any other valid reason the developer did not adhere to the guideline.

By doing so, they can more easily avoid security jargon and speak in clear, mutually understood language. They will talk about a *guideline violation* instead of a *vulnerability*, *exploit*, or a *security defect*. Terminology with subtle differences in meaning that are likely not fully understood by developers. There is no need to talk about vulnerability types or use acronyms, instead the focus can be on the desired result, spoken in development terms. So instead of warning of a potential Cross-Site Scripting (XSS) vulnerability, the security team can indicate a lack of output escaping and request the use, or development, of a library for this purpose.

By improving culture, and creating more empathy between the security team and developers, the security team should be better equipped to integrate security and development workflows without hurting productivity. The integration of these processes can be facilitated with appropriate technology that is more suited to developers. Instead of forcing re-purposed security tools designed for security professionals onto developers, new technology should be used that is built with the developer experience in mind.

1.3 Developer-minded security education

Deliberate security education that keeps the developer experience in mind should be:

- *relevant* to the developer's work,

- *efficient* in achieving the developer's needs,
- *usable*, engaging, fun.

The goal of security education is to teach developers the knowledge and skills necessary to produce secure code. To achieve this goal, we want the acquired information to be stored in long-term memory. Every piece of knowledge in long-term memory is stored as a series of associations [9].

If a developer learns about the programming language Kotlin, this can be encoded in their memory under the following associations:

- Programming languages used for Android apps.
- Programming languages designed by JetBrains.
- Things I learned about while eating pizza.

More associations, and more meaningful associations, make it easier to retrieve information from memory. Education should be designed to allow for many meaningful associations. We often make numerous unconscious associations by utilizing all of our senses, such as the association between Kotlin and pizza in the example above. While these associations may seem random, they are still used to retrieve information and can even be used to design better education [9].

1.3.1 Relevant education

Some of the associations made while learning secure coding skills will be related to the practical context in which the developer is taught. This context should resemble the one where the acquired skills are applied, as this will improve retrieval of the content. This phenomenon is one of the reasons why pilots do not learn to fly a plane through slide presentations but by using flight simulators [9]. Similarly, developers should not learn secure coding through books or slide presentations.

The learning context should resemble the developer's work context. They should receive education in their office or home office, by using their own computer, their own keyboard and mouse, and through actual code. This code should be in the same programming language and framework they are using daily. Even the type of software should be relevant to the developer's work. A developer working on mobile applications should be taught secure coding by means of code for mobile applications.

1.3.2 Efficient education

Retention and recollection of the material is not just improved through more and better associations, but also through repetition. Repetition reinforces the associations in memory, this makes them stronger and more durable [9]. Of course, we can not expect developers to study for hours on a daily basis, education should be possible with minimal harm to their productivity.

Finding the right balance between repetition and efficiency is a difficult problem. How much repetition is needed depends on the individual learning pace of each developer. A teacher can easily adapt their pace to the students in their classroom. However, in classroom teaching, much of the practical context and other relevant aspects mentioned in the previous section are easily neglected. In online learning it is more practical to allow each developer to train using their own programming language and their own machine. Online learning also allows for better scalability. But online systems are worse at adapting to the learning pace of each user. A possible solution to this problem is described in Part I of this work.

From the developer's perspective, relevance and efficiency are closely related. As explained in the previous section, education in the right language and framework allows for better recollection because contextual associations are made in memory. But using the right programming language also ensures that the developer is being taught problems and solutions that are immediately applicable to their work. Many learning resources teach secure coding concepts in a different programming language, or by using pseudo-code. While most developers will be able to apply the learned concepts to their own code, they will still need to research specifics on their own to do so.

Efficiency can also be improved by providing exercises that help developers acquire the right skills. These skills are recognizing insecure code patterns and (re)writing their code so that it is secure. They need to be taught the *paved path*, and how to stay on it. Security experts, on the other hand, are often occupied with testing whether potential vulnerabilities can be exploited. This is frequently reflected in security education; developers are handed penetration testing exercises. Such exercises certainly have a place in developer education, as they create new and strengthen old associations in memory. However, in order to teach developers secure coding skills more efficiently, the focus should be on defensive exercises, i.e., exercises that teach the developer to recognize insecure code and to prevent or fix insecurities.

1.3.3 Usable education

When it comes to online learning, usability and engagement go hand in hand. By improving the usability of the education, engagement is increased as well [10–13]. Engagement has been shown to have a clear positive effect on learning [13, 14]. Because online learning often suffers from low engagement, extra care should be put into usability [15].

Improving the relevance and efficiency of education will indirectly make it more usable, as less effort will be needed to understand and apply the learned lessons. Efficient training, that can avoid unnecessary repetition, will make it less likely that a developer gets bored due to this repetition. On the other hand, too much efficiency can cause the learning curve to be unnecessarily steep. When developers have difficulty keeping up with the material, they might experience frustration. A bored or frustrated developer is likely to lose interest and even to disengage from the education. When instead a developer continuously experiences the right level of challenge, they are experiencing a state of flow [16, 17]. Flow affects learning both directly and via increased engagement [14, 16, 18]. Finding the right difficulty for each user to keep them in a state of flow is a challenge tackled in Part I of this work.

Besides keeping users in a state of flow, usability and engagement can also be improved through other means. One way to do this is by providing a structured learning journey that lays out a clear path and expectations [15]. This structured journey allows online learning to replicate a more traditional learning experience. Another way to improve usability and engagement is through increased interactivity of the education [15]. It is possible to encourage interactivity between learners by adding gamification and competitive aspects such as high scores, leaderboards, badges, achievements, or even prizes. Promising rewards like these upon completion of certain tasks has a clear effect on learner motivation and hence engagement. Finally, engagement can also be increased through the presentation of the learning material. Many developers are problem solvers at heart. They enjoy trying to understand a problem and coming up with an elegant solution. Rather than handing them descriptive learning material, it will likely improve engagement when developers are allowed to figure out the answer for themselves. In online learning this is not easily achieved, since automated grading of open ended questions is difficult.

A form factor that still allows for some type of problem solving that is easy to correct for a computer is multiple choice. This type of questions teaches a developer to *recognize* the right answer among several [9]. In reality, a developer also needs to *recall* the material without being given options to choose from. However, such a recollection activity can be turned into a recognition activity by providing so-called scaffolding [19]. In the next section I explain how security tools can be designed to provide this scaffolding during development.

1.4 Developer-minded security tools

Education alone is insufficient for developers to produce secure code. Our memories are not infallible and regression of knowledge is possible. Other times developers sufficiently remember, but fail to apply their knowledge in practice [20]. Security tools should help close this gap between knowledge and practice.

Developers are known to dislike and often disable security tools during development [6]. They frequently perceive them as one of the biggest inhibitors of productivity. A tool supporting the paved path methodology should in the first place be designed as a developer tool, security should come as an indirect result.

A developer tool should:

- provide *relevant* feedback to the developer's work,
- be *efficient* and improve productivity instead of hurting it,
- be *usable* and well integrated into developer workflows.

To the developer, the goal of a tool is to make development easier. It should help boost their productivity and lower their cognitive burden. From the developer's perspective, security is a non-functional requirement and not the main objective. By using the paved path methodology, such a developer tool can improve the security of the code. Good development teams are narrow in their allowed practices, as this makes it more feasible to understand and maintain the software. In order to be successful, the security team should contribute and help decide these allowed practices. Security will then simply be a result of sticking to these practices, a result of following the path of least resistance.

1.4.1 Relevant tools

To lay out this paved path, the security team and the development team have to work closely together. Together, they need to create guidelines that specify the preferred solution for a security-critical feature that is needed by the development team. These guidelines should not be conceptual guidelines like the security team is used to creating, as those are hard to translate into code by developers. Instead, the security team should work together with members from the development team to make specific, Application Programming Interface (API)-level guidelines. These guidelines lay out which libraries or even which specific library calls are to be used in the project.

When the developers are using libraries for a specific purpose, they are focused on the functionality of their code. Security is often orthogonal to that purpose. When a library is used insecurely by a developer, we should not blame the developer for this, but instead blame the design of the library. When laying out the paved path, it is important that no security bugs can be introduced by *using* the chosen library, and instead all possible bugs are contained within the *implementation* of the library itself. Custom (wrapper) libraries may need to be developed that are inherently safe and that can be freely used by the developers.

A tool supporting the paved path methodology should then remind the developer of the agreed guidelines any time they stray from the paved path. Any library calls or custom methods that the developer uses and that are functionally similar to the library provided by the security team, should be marked by the tool. This should be done regardless of the security of the used library. The goal of the tool is not to evaluate the security of the code, but only its adherence to the guidelines. Since these guidelines are customized for each project, this guarantees that the feedback will always be applicable and highly relevant to the developer's work. The tool should hence be easy to configure so that project-specific guidelines can be enforced.

1.4.2 Efficient tools

It is worth emphasizing that the enforced guidelines can have a wide range of applications. They can be used to migrate to a new library, to deprecate old functions, enforce code quality guidelines, improve legibility, and so on. Such a tool helps developers share their knowledge and guide each other to improve the quality and maintainability of the software.

To assert that the developer is adhering to the guidelines, only local analyses are required. There is no need for complex data flow or control flow analyses. The required local analyses can be done in real time, as the developer types. A tool that supports the paved path methodology is hence efficient and actively improves the productivity of the developer instead of hurting it.

In contrast, traditional security tools will try to assert the absence of certain bugs in the application. To do this, they have to analyse all possible data flows. Even moderately complex applications contain complex data flows that are difficult to reason about [21]. This complexity leads to slow scanning speed and, to the developer's standards, insufficient quality of results.

When using traditional security tools, developers report this poor quality of feedback and slow scanning speed as big inhibitors of their productivity [6]. These security tools are of course still valuable to the security team to help them create and maintain libraries, but they are not the right tool to provide to developers.

Because the tool is able to provide instantaneous feedback, it makes sense to do so in the developer's Integrated Development Environment (IDE). It can then make all necessary information about the agreed guidelines available in the IDE as well. This way, the developer is able to access it without making a context switch to consult outside documentation, again boosting productivity.

The guidelines describe which library calls to use, that is, they describe the desired outcome. Because of this, the tool that is enforcing the guidelines is able to provide targeted and relevant remediation guidance that can even be automated by the tool.

The lack of such remediation guidance is a frequently mentioned inhibitor of productivity in traditional security tools [6] which usually can only provide generally applicable guidance [22].

1.4.3 Usable tools

Avoiding the need for research does not only boost productivity, it also makes the feedback easier to understand. Because the information is presented as a guideline, it is immediately obvious for the developers what is expected of them to fix the problem. Adhering to a coding guideline is trivial, and does not require knowledge of the possible vulnerabilities it mitigates.

When the developer is provided feedback that describes potential vulnerabilities, on the other hand, solving the problem is a more complex task. Even if the developer is already familiar with the vulnerability and does not need to research it, the focus in the case of a potential bug report is on determining whether this particular instance can be exploited or not. If it is determined insecure, the developer needs to research a solution, apply it, and verify whether no functional changes were made. This requires a larger cognitive effort. Traditional security tools are meant to be just that, tools for the security team.

The usability for developers is improved because the tool is designed with the developer in mind. The tool resides in the developer's IDE and reuses existing features to display information and provide remediation guidance. It is closely integrated in the developer's workflow and is easily and frequently used for other purposes than security.

1.5 This book

In Part I of this work, I describe how to create better security education and keep the developer experience in mind. I analyzed the behaviour of over 175,000 developers receiving security education on the online learning platform created by SCW. I designed, implemented, and evaluated an Intelligent Tutoring System (ITS) that selects the most appropriate exercise for each developer at each point in time. This ITS ensures that developers are receiving *relevant* training and acquire the necessary skills *efficiently* while still being *usable* and engaging. The research in this part is based on a large data set made up of data from developers at large corporations. This data is rigorously analyzed and the results are strong evidence towards the proposed solutions.

This part is based on:

- **Method and System for Adaptive Security Guidance**
  Pieter De Cremer, Matias Madou, Nathan Desmet, Colin Wong
  US Patent Application 16/234,037, 2018
  US Patent Publication US20200211135A1, 2020

- **Create a Certification Framework for Secure Development Practices**
  Matias Madou, Brian Chess, Pieter De Cremer
  Enhancing Software Supply Chain Security, NIST, 2021

In Part II, I describe the goals and requirements for a security tool supporting the paved path methodology. I helped with the design and requirements of this tool that is implemented by the engineering team at SCW and evaluated it both in a controlled experiment as well as in professional settings. The tool, called Sensei, functions as an IDE plugin and reuses existing IDE features that the developer is familiar with. It provides remediation guidance in the form of quick-fixes and offers several features improving usability for the developer. Sensei is designed as a developer tool first, improving productivity and reducing the cognitive burden of development. It enables close collaboration between the development team and the security team, and ensures that secure code is being produced in a scalable way. The research in this part is mostly based on observations at companies and second hand information acquired through interviews. Since the research subjects are developers working for paying customers at Secure Code Warrior, they were not able to invest large amounts of time to collaborate in this research. The results from these interviews do not offer a positive and final proof for the hypothesis of this work due to the designed experiments carried out with less than expected rigorousness, therefore, they can not be interpreted as strong evidence towards the proposed solution. Nonetheless, they serve as valuable informational insights in further improvements of the tool and for future experiments, which I explain in detail in Section 9.2.3.

This part is based on:

- **Sensei: Enforcing secure coding guidelines in the integrated development environment**
  Pieter De Cremer, Nathan Desmet, Matias Madou, Bjorn De Sutter
  In *Software: Practice and Experience*, 2020

- **Method and Apparatus for Detecting and Remediating Security Vulnerabilities in Computer Readable Code**
  Pieter De Cremer, Matias Madou, Nathan Desmet, Colin Wong
  US Patent Application 17/005,685, 2020

- **Method and Apparatus for Generating Security Vulnerability Guidelines**
  Pieter De Cremer, Matias Madou, Nathan Desmet, Colin Wong
  US Patent Application 17/005,729, 2020

- **Promote a Paved Path Secure Development Methodology**
  Matias Madou, Brian Chess, Pieter De Cremer
  Enhancing Software Supply Chain Security, NIST, 2021

Finally, in Part III, I describe related tools and practices, and offer my opinion on how they can be used to achieve a more human-centered approach to software security. Some of the discussed tools and practices have been thoroughly researched and are used in industry, other work is more recent and innovative. This includes practices around governance, training, development, building, and deploying of software products.

Other contributions as a researcher at SCW that are not detailed in this dissertation include:

- **Actionable Software Security for Developers**
  IWT bedrijfssteun, 2015

As part of this project I surveyed and researched security problems introduced in Java applications by incorrect use of common APIs. Out of this research I created close to 100 rules that can be enforced in the Sensei IDE plugin.

- **How to scale application security training for developers**
  VLAIO O&O project, 2017
  As part of this project I surveyed and researched common security problems introduced in mobile applications created using the Android API. From this research, I created a mobile application and introduced vulnerabilities into it to create 138 exercises on the SCW training platform.

- **Aanpasbare ondersteuning voor veilige software ontwikkeling**
  VLAIO O&O project, 2019
  I invented and described automatic rule creation methods for the Sensei IDE plugin to be developed with this funding. The methods include:
  - Improved manual creation of rules through rule editor
  - Static generation of rules based on static analysis tool results
  - Static generation of rules based on code repository history
  - Dynamic generation of rules based on developer behaviour

- **Secure Code Bootcamp**
  European Social Fund (ESF) project, 2020
  I was part of the design and product management of a mobile application to provide secure coding education to developers. This application is now freely available in the Android Play store and the iOS App store.

1.6 Publication output

My publication output during my research includes one journal paper, two position papers, and four patents.

- **Sensei: Enforcing secure coding guidelines in the integrated development environment**
  Pieter De Cremer, Nathan Desmet, Matias Madou, Bjorn De Sutter
  Journal paper, peer reviewed and published [23].
  In *Software: Practice and Experience*, 2020

- **Create a Certification Framework for Secure Development Practices**
  Matias Madou, Brian Chess, Pieter De Cremer
  Position paper, peer reviewed and published [24].
  Enhancing Software Supply Chain Security, NIST, 2021

- **Promote a Paved Path Secure Development Methodology**
  Matias Madou, Brian Chess, Pieter De Cremer
  Position paper, peer reviewed and published [24].
  Enhancing Software Supply Chain Security, NIST, 2021

- **Method and System for Adaptive Security Guidance**
  Pieter De Cremer, Matias Madou, Nathan Desmet, Colin Wong
  Patent application, reviewed and granted [25].
  US Patent Application 16/234,037, 2018
  US Patent Publication US20200211135A1, 2020

- **Method and Apparatus for Detecting and Remediating Security Vulnerabilities in Computer Readable Code**
  Pieter De Cremer, Matias Madou, Nathan Desmet, Colin Wong
  Patent application, under review with the examiner.
  US Patent Application 17/005,685, 2020

- **Method and Apparatus for Generating Security Vulnerability Guidelines**
  Pieter De Cremer, Matias Madou, Nathan Desmet, Colin Wong
  Patent application, under review with the examiner.
  US Patent Application 17/005,729, 2020

- **Method and Apparatus for Adaptive Security Guidance**
  Pieter De Cremer, Matias Madou, Nathan Desmet, Colin Wong
  Patent continuation application, under review with the examiner.
  US Patent Application 17/469,636, 2021

1.7 Perspectives

There is still progress to be made outside of this work, both in the education and tools for the paved path methodology as well as in different aspects of the cooperation between the security team and the development team.

Until now, education and tools were considered two separate things to provide to developers. In reality, the border between these two is not that clearly defined and they blend over into each other.

Developers often learn while doing, and the educational aspect of Sensei itself should not be underestimated. The approach of an ITS can be extended to the tool. The information, and even guidance, that is provided by Sensei should not be identical for each user. Different information should be offered when teaching a new lesson compared to brushing up a forgotten one. Teaching new concepts should also depend on previous knowledge. It is easier to explain a new type of injection flaw (e.g. Extensible Markup Language (XML) injection) to a developer by drawing parallels with other injection flaws that they are already acquainted with (e.g. Structured Query Language (SQL) injection). Markings and remediation guidance should depend on the knowledge of the developer, and even that of other team members. Some critical security features might need more expertise, and the most efficient way forward as a team, could be for an uninformed developer to ask help from a more educated or experienced coworker.

But not only Sensei can benefit from this blended border with education. Education can also be more targeted if it can integrate with Sensei and other developer tools. By observing which errors developers make in practice, a better picture of their understanding can be created. By integrating with issue trackers, it is possible to keep track of the issues that developers are assigned to, and to design individual learning goals for each developer. Integration with developer tools enables education to become even more relevant, efficient, and usable.

Another challenge that remains for the paved path methodology is applying it to existing, potentially large, legacy codebases. While the security team can still lay out a paved path for developers to follow, it is no easy task to refactor the existing code so that it adheres to this path. Extra care should be taken when designing the inherently safe wrapper library so that its uses match one to one as much as possible with the existing library.

The paved path methodology helps the security team prevent and fix vulnerabilities at scale. However, the security team still needs to tackle scalability in other facets of their collaboration with developers beyond the scope of this work. One example is threat modeling. It is impossible for one security expert to model threats for all software designs by up to 200 developers. Even though some approaches exist, they need wider adoption and more thorough evaluation before they can be endorsed.

The paved path methodology by itself will not cause an enormous shift in software development, nor will it single-handedly prevent all software vulnerabilities in the future. But it is an improvement. With this method, we are paving the path towards secure development, and taking a step in the right direction. A step towards a more human-centered future of software security, in a bigger journey to make security a shared responsibility among everyone involved in the software development process.

Part I
Education

Pass on what you have learned. Strength. Mastery. But weakness, folly, failure also. Yes, failure most of all. The greatest teacher, failure is.

Yoda

Introduction

Education in the paved path methodology should be deliberate and keep the developer experience in mind. The SCW training platform is a good resource for education in the paved path methodology. It provides online training through defensive secure coding exercises in many different programming languages and frameworks, and its gamification and interactivity make it fun and usable. Despite the focus on developers and its many usability features, there is still a significant part of the user base that only follows a minimal amount of training. Users follow one of the predetermined courses, and it is likely that the pacing of these courses does not fit their needs. Users get bored due to too much repetition, or frustrated because the content is moving too fast.

I created an ITS by combining psychometric models with techniques from computer science to recommend exercises to each individual at any point in time. A customized recommendation like this is more likely to keep the developer engaged and allows meaningful learning to take place.

I start Part I by giving an overview of the SCW training platform, its features, and the different types of exercises in Chapter 2. I then describe the design of the ITS and explain the used techniques in more detail in Chapter 3. The last chapter in this part provides an evaluation of the ITS before offering some perspectives that remain future work.

Chapter 2

Secure Code Warrior

In the paved path methodology, developers should be provided with deliberate, targeted education that keeps the developer experience in mind. This education should be *relevant* to the developers work, *efficient* in achieving their needs, and *usable* to keep them engaged.

In this chapter, I describe the education provided by the online learning platform created by Secure Code Warrior (SCW). I assess its potential for use in the paved path methodology and describe shortcomings that require further research.

If nothing else, take away from this chapter...

The SCW training platform provides training to hundreds of thousands of developers from reputable customers. It provides defensive exercises in a gamified and engaging way and offers a wide variety of programming languages and frameworks. It is suitable to be used in the paved path methodology as it is *relevant* and *usable*. However, there is room for improvement when it comes to the *efficiency* of the training. All users, regardless of skill level, are presented challenges of the same difficulty. This leads to boredom or frustration for some users and might cause them to disengage from the training.

2.1 The company

The company was co-founded by Pieter Danhieux and Matias Madou Ph.D., two alumni of Ghent University and both globally recognized security experts. During their international careers, both founders noticed that the focus in industry is too often on remediation rather than prevention of security problems in software. Their vision is not to make a security expert out of every developer, but to empower them to become the first line of defence in the organisation. The company provides education and tools to improve secure coding skills of developers. Both the online training platform and their IDE based security tool can be deployed to support a paved path methodology as described in this book.

Since its start in 2015, more than 400 customers from almost 40 countries around the world use SCW products to improve the secure coding skills within their development teams. SCW focuses on large companies with lots of developers. Most customers are active in banking, finance, government, aviation, or telecommunications. Some notable customers are:

- Coupang: the largest online retailer in South Korea
- 19 of the top 100 global banks
- The British Broadcasting Corporation (BBC): the largest broadcaster in the world
- 2 of the world's largest telecommunications providers
- 2 of the top US credit card processors
- Zoom: one of the largest communication technology companies

2.2 The training platform

The training platform provides an interactive and gamified way to learn secure coding concepts, and focuses on defensive techniques. In the mission control dashboard, shown in Figure 2.1, developers are tasked with defending an application from different types of threats originating from all over the world. The developer is awarded points for completing exercises, and leaderboards are shown to create a competitive environment. By collecting enough points and spending enough time on the platform, the developer can unlock achievements and gain badges. All of this progress can be monitored on a metrics dashboard, shown in Figure 2.2. A total of over 100,000 unique developers used the training platform in 2020.

In a survey with 722 developers, 90% of respondents said they prefer SCW over traditional classroom learning and 85% prefer it over other online learning resources they have tried in the past. These results are also confirmed by many testimonies, such as the following response on TechValidate[1].

> Secure Code Warrior's use of gamification has helped us emphasize the importance of secure coding in a refreshingly fun and engaging way. – *Developer at Global 500 Financial Services Company*

2.3 Exercises

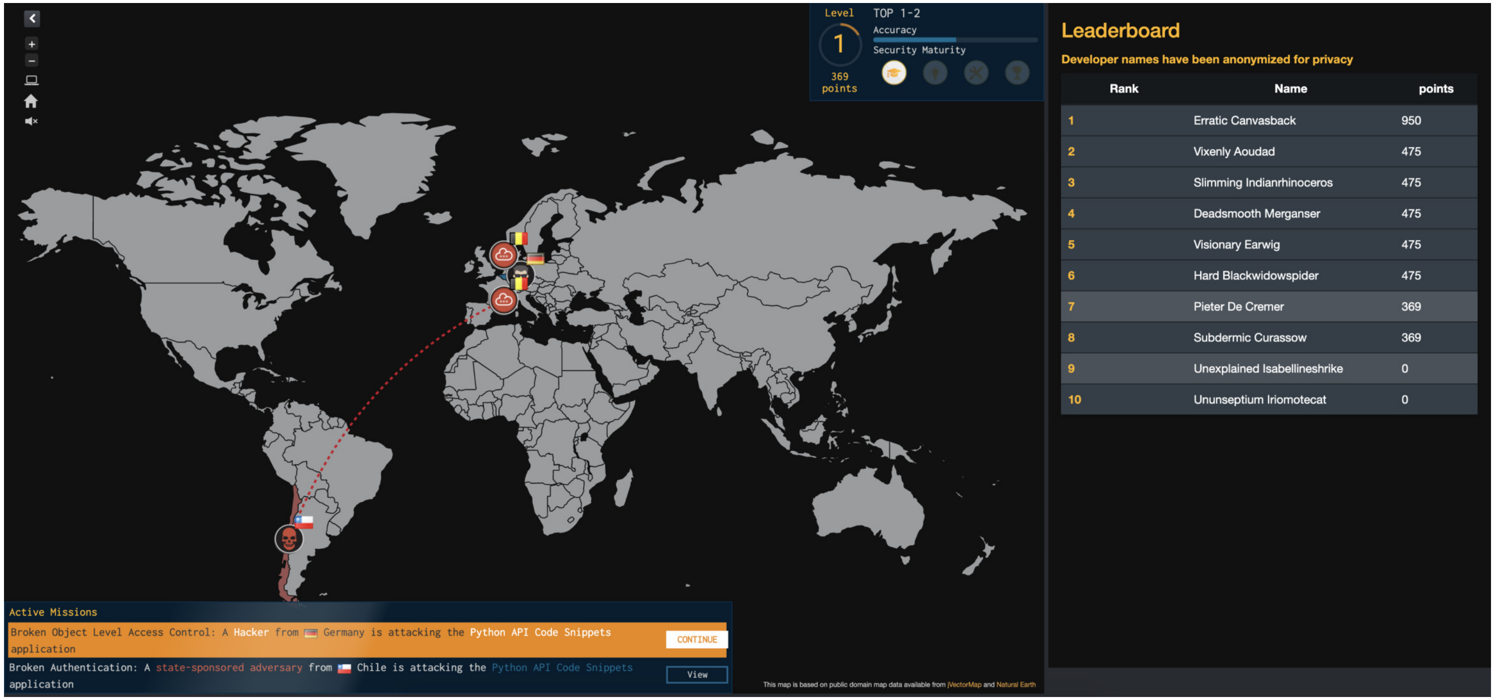
Training exercises on the SCW platform, often called challenges, are most frequently created from a complete and secure software application such as a webstore or a banking application. To create a challenge, a vulnerability is introduced into this application on purpose. The challenge is presented to the users as one of three types of exercises, each assigned a numerical level, an *identify* (L1), *locate* (L2), or *fix* exercise (L3).

Identify exercises (L1) mark the insecure code fragment and provide the developer with a number of vulnerability categories. It is up to the developer to identify which of the provided categories best describes the vulnerability present in the code fragment. In Figure 2.3 an identify exercise is shown based on a Structured Query Language (SQL) injection in a Python web application.

For locate exercises (L2), the category of the vulnerability that is present in the insecure code fragment is given. The insecure code fragment is marked, as well as several other (secure) code fragments. It is up to the developer to locate which code fragment contains the insecurity. An example of a locate exercise is shown in Figure 2.4 using the same SQL injection in the same Python web application as the identify exercise in Figure 2.3.

Fix exercises (L3) show both the insecure code fragment and the category of the inserted vulnerability to the user. Four alternatives are shown, with changes made to the insecure code fragment, and sometimes to other parts of the application code as well. The developer needs to find the most secure alternative among the four options. A fix exercise is shown in Figure 2.5, again using the same SQL injection as before.

[1] `techvalidate.com/product-research/secure-code-warrior/facts`



Level
1
369 points
TOP 1-2
Accuracy
Security Maturity
CONTINUE
View
This map is based on public domain map data available from jVectorMap and Natural Earth

Leaderboard

Developer names have been anonymized for privacy

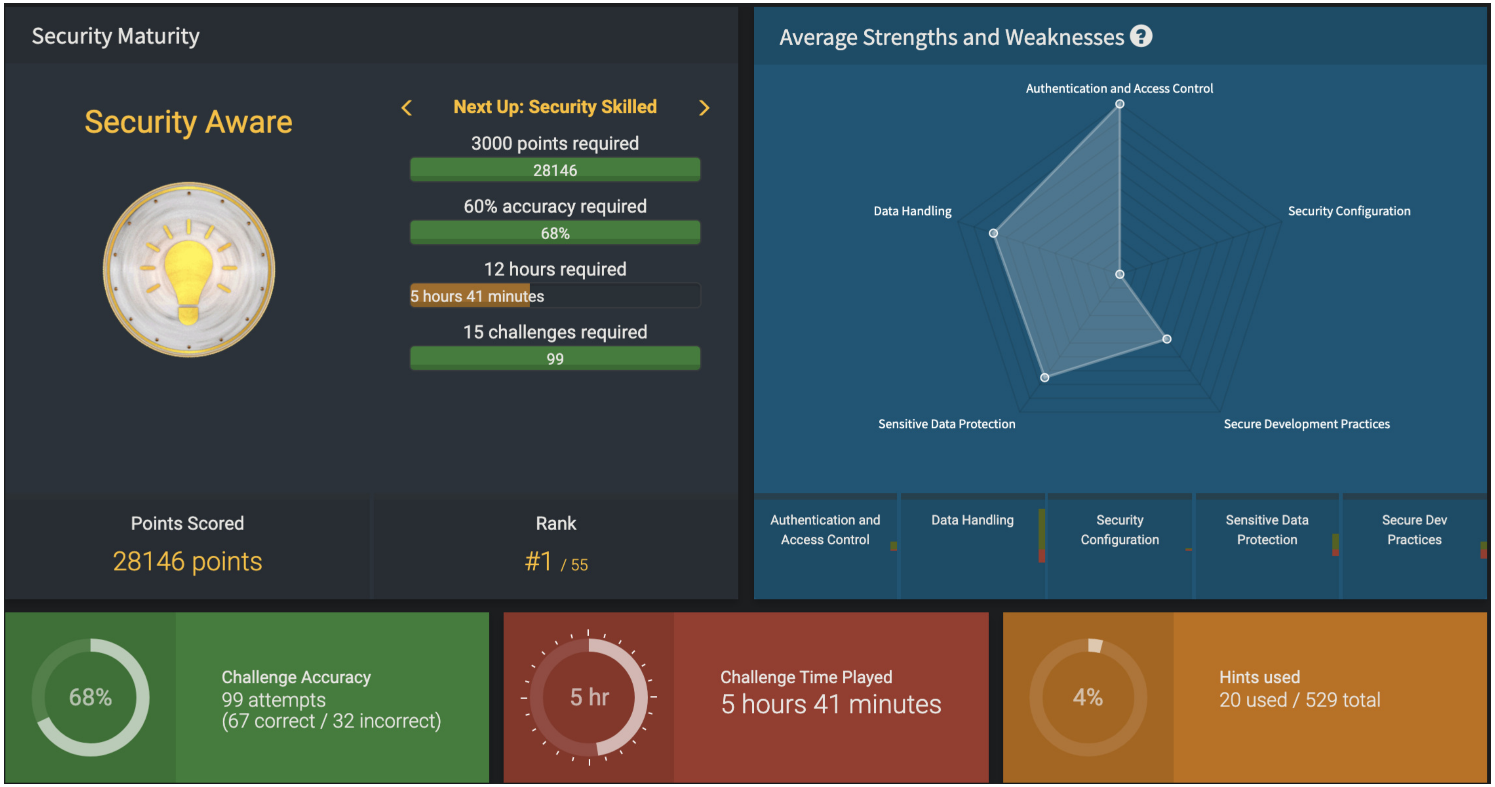
Rank	Name	points
1	Erratic Canvasback	950
2	Vixenly Aoudad	475
3	Slimming Indianrhinoceros	475
4	Deadsmooth Merganser	475
5	Visionary Earwig	475
6	Hard Blackwidowspider	475
7	Pieter De Cremer	369
8	Subdermic Curassow	369
9	Unexplained Isabellineshrike	0
10	Ununseptium Iriomotecat	0

Active Missions

Broken Object Level Access Control: A Hacker from Germany is attacking the Python API Code Snippets application

Broken Authentication: A state-sponsored adversary from Chile is attacking the Python API Code Snippets application

Figure 2.1: The mission control dashboard on the SCW platform creates a gamified overview of the exercises.


Security Maturity

Security Aware

Next Up: Security Skilled

3000 points required
28146

60% accuracy required
68%

12 hours required
5 hours 41 minutes

15 challenges required
99

Points Scored
28146 points

Rank
#1 / 55

Average Strengths and Weaknesses

Authentication and Access Control
Security Configuration
Secure Development Practices
Sensitive Data Protection
Data Handling

Authentication and Access Control | Data Handling | Security Configuration | Sensitive Data Protection | Secure Dev Practices

68%
Challenge Accuracy
99 attempts
(67 correct / 32 incorrect)

5 hr
Challenge Time Played
5 hours 41 minutes

4%
Hints used
20 used / 529 total

Figure 2.2: The metrics dashboard on the SCW platform allows developers to monitor their progress and unlock new badges.

Fix (L3) exercises are often combined with identify (L1) or locate (L2) exercises. These challenges then consist of two stages, in the first stage the vulnerability needs to be identified or located, in the second stage the exact same vulnerability needs to be fixed. The resulting two-stage challenge is an identify-and-fix (L4 = L1 + L3) or a locate-and-fix (L5 = L2 + L3) challenge.

Recently new and more interactive challenge types are being developed. One such type requires the developer to construct input that successfully exploits the vulnerability present in the application.

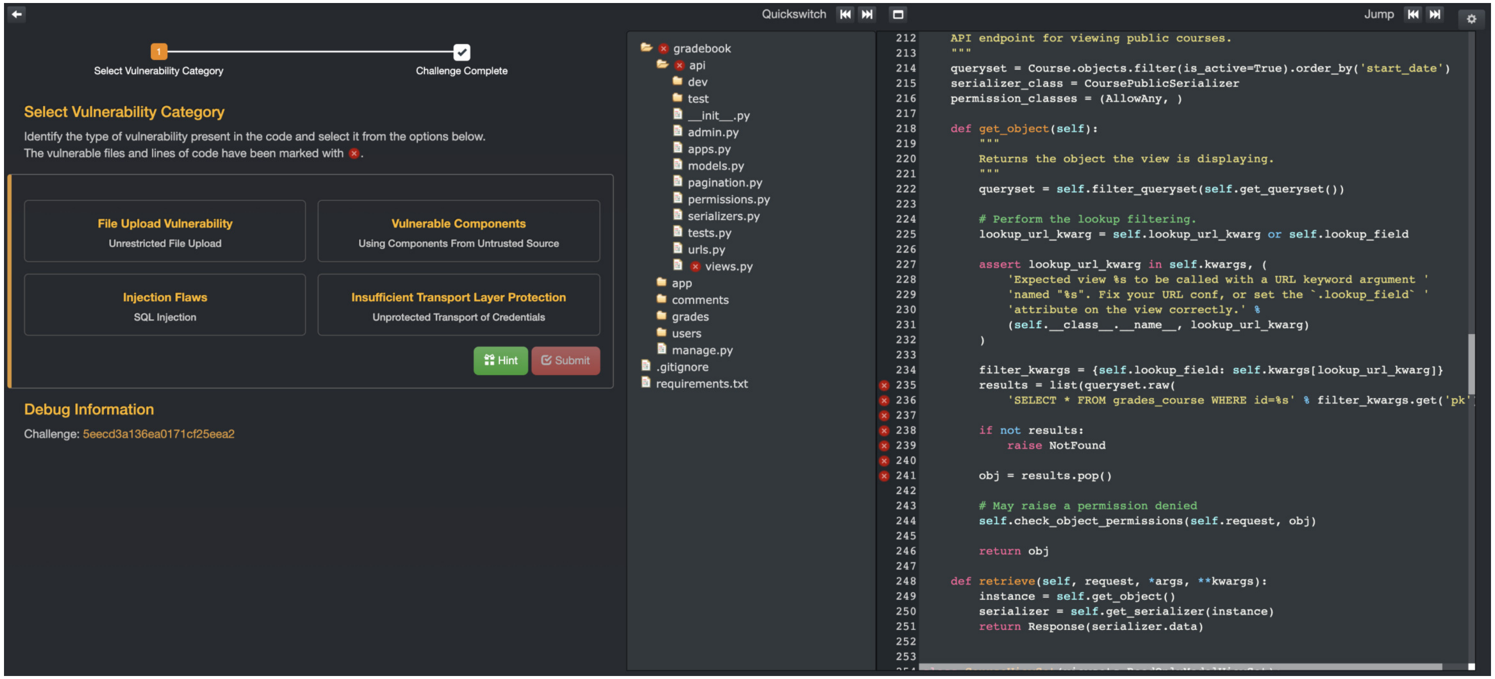
2.4 Context

The challenges are presented to users in different contexts, these are training mode, tournament mode, or assessment mode.

The default context is the training mode. In this mode the developer is allowed to use as many hints as needed. They are also allowed an unlimited amount of attempts to find the right answer. Each hint or failed attempt reduces the amount of points that the developer is awarded. Appendix A describes in more detail how the difficulty of a challenge, the amount of hints used, and the number of failed attempts determine how many points are awarded. Developers are free to choose which challenges they solve first in training. A standard course that guides them through the Open Web Application Security Project (OWASP) Top 10 categories is provided, and many developers complete this course before trying other challenges.

In tournament mode, the scoring method and availability of hints and attempts can be adjusted by the host. In this mode, all participants are shown exercises about the same vulnerability type and of the same difficulty but in their language of choice. The tournament is run for a limited time, usually a few hours, in which the contestants can complete the challenges. In tournament mode, a live leaderboard is visible that can optionally be hidden close to the end for suspense. Since all participants are shown the same number of exercises having the same difficulty, it often comes down to speed to finish the challenges in time, and accuracy to lose as few points as possible through hints or mistakes.

In assessment mode no hints are available and only one attempt is allowed for each challenge. This mode is used to evaluate the performance of a user. Customers can select the vulnerability type and difficulty of the challenges making up the assessment. There are some templates provided as an example that test for knowledge of the OWASP Top 10.



Select Vulnerability Category

Identify the type of vulnerability present in the code and select it from the options below.
The vulnerable files and lines of code have been marked with ✖.

File Upload Vulnerability Unrestricted File Upload	**Vulnerable Components** Using Components From Untrusted Source
Injection Flaws SQL Injection	**Insufficient Transport Layer Protection** Unprotected Transport of Credentials

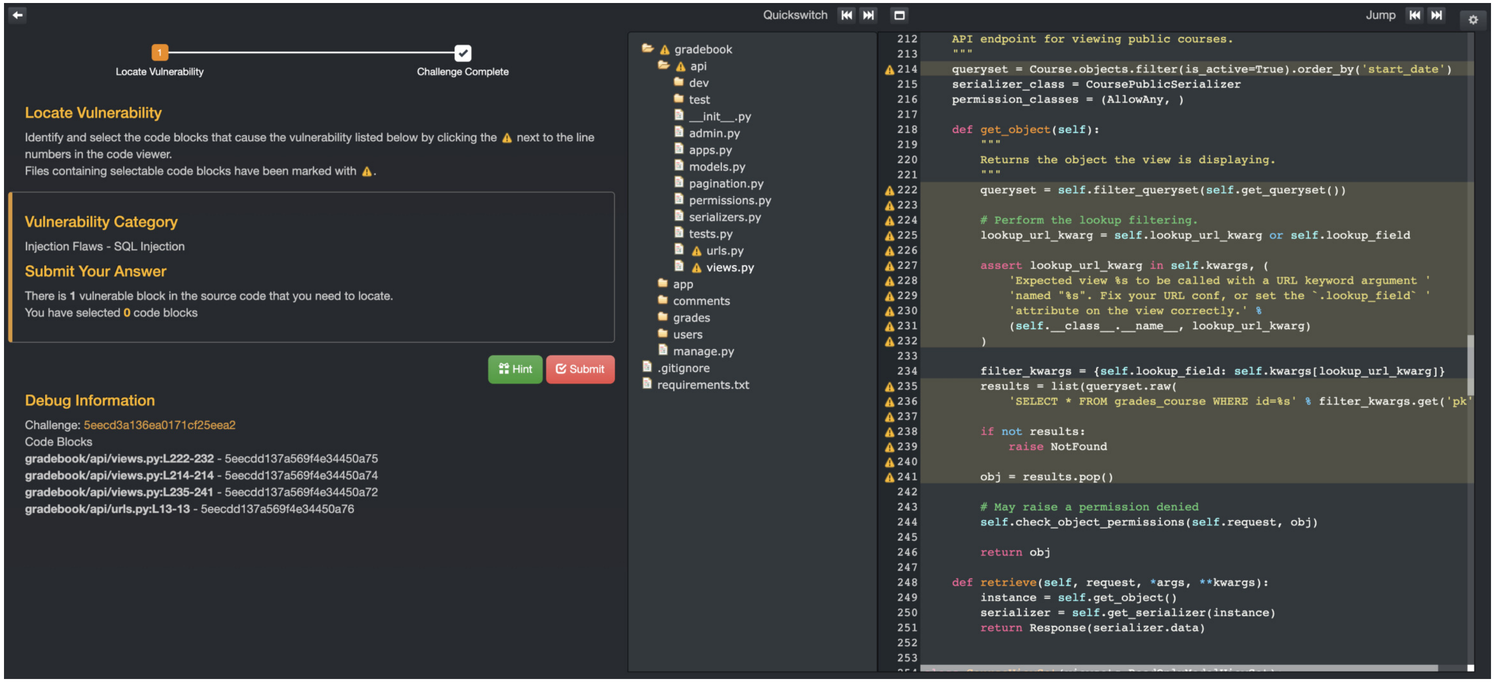
Debug Information

Challenge: 5eecd3a136ea0171cf25eea2

```
✖ gradebook
    ✖ api
        dev
        test
        __init__.py
        admin.py
        apps.py
        models.py
        pagination.py
        permissions.py
        serializers.py
        tests.py
        urls.py
        ✖ views.py
    app
    comments
    grades
    users
    manage.py
.gitignore
requirements.txt
```

```
API endpoint for viewing public courses.
"""
queryset = Course.objects.filter(is_active=True).order_by('start_date')
serializer_class = CoursePublicSerializer
permission_classes = (AllowAny, )

def get_object(self):
    """
    Returns the object the view is displaying.
    """
    queryset = self.filter_queryset(self.get_queryset())

    # Perform the lookup filtering.
    lookup_url_kwarg = self.lookup_url_kwarg or self.lookup_field

    assert lookup_url_kwarg in self.kwargs, (
        'Expected view %s to be called with a URL keyword argument '
        'named "%s". Fix your URL conf, or set the `.lookup_field` '
        'attribute on the view correctly.' %
        (self.__class__.__name__, lookup_url_kwarg)
    )

    filter_kwargs = {self.lookup_field: self.kwargs[lookup_url_kwarg]}
    results = list(queryset.raw(
        'SELECT * FROM grades_course WHERE id=%s' % filter_kwargs.get('pk'

    if not results:
        raise NotFound

    obj = results.pop()

    # May raise a permission denied
    self.check_object_permissions(self.request, obj)

    return obj

def retrieve(self, request, *args, **kwargs):
    instance = self.get_object()
    serializer = self.get_serializer(instance)
    return Response(serializer.data)
```

Figure 2.3: SQL injection in a Python web application presented as an identify exercise, the first of three different challenge types on the SCW platform.



Locate Vulnerability

Identify and select the code blocks that cause the vulnerability listed below by clicking the ⚠ next to the line numbers in the code viewer.
Files containing selectable code blocks have been marked with ⚠.

Vulnerability Category

Injection Flaws - SQL Injection

Submit Your Answer

There is **1** vulnerable block in the source code that you need to locate.
You have selected **0** code blocks

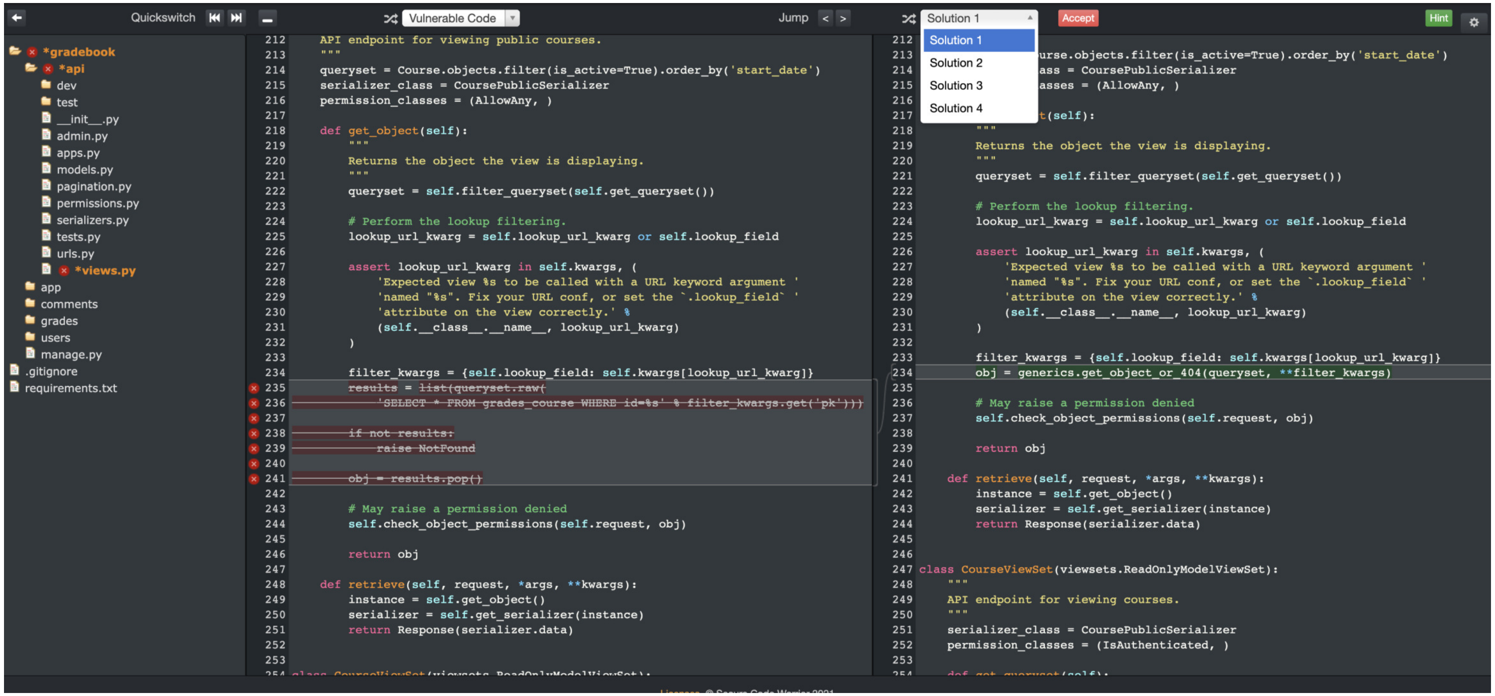
Debug Information

Challenge: 5eecd3a136ea0171cf25eea2
Code Blocks
gradebook/api/views.py:L222-232 - 5eecdd137a569f4e34450a75
gradebook/api/views.py:L214-214 - 5eecdd137a569f4e34450a74
gradebook/api/views.py:L235-241 - 5eecdd137a569f4e34450a72
gradebook/api/urls.py:L13-13 - 5eecdd137a569f4e34450a76

```
⚠ gradebook
  ⚠ api
      dev
      test
      __init__.py
      admin.py
      apps.py
      models.py
      pagination.py
      permissions.py
      serializers.py
      tests.py
      ⚠ urls.py
      ⚠ views.py
    app
    comments
    grades
    users
    manage.py
  .gitignore
  requirements.txt
```

```
  212      API endpoint for viewing public courses.
  213      """
⚠ 214      queryset = Course.objects.filter(is_active=True).order_by('start_date')
  215      serializer_class = CoursePublicSerializer
  216      permission_classes = (AllowAny, )
  217
  218      def get_object(self):
  219          """
  220          Returns the object the view is displaying.
  221          """
⚠ 222          queryset = self.filter_queryset(self.get_queryset())
⚠ 223
⚠ 224          # Perform the lookup filtering.
⚠ 225          lookup_url_kwarg = self.lookup_url_kwarg or self.lookup_field
⚠ 226
⚠ 227          assert lookup_url_kwarg in self.kwargs, (
⚠ 228              'Expected view %s to be called with a URL keyword argument '
⚠ 229              'named "%s". Fix your URL conf, or set the `.lookup_field` '
⚠ 230              'attribute on the view correctly.' %
⚠ 231              (self.__class__.__name__, lookup_url_kwarg)
⚠ 232          )
  233
  234          filter_kwargs = {self.lookup_field: self.kwargs[lookup_url_kwarg]}
⚠ 235          results = list(queryset.raw(
⚠ 236              'SELECT * FROM grades_course WHERE id=%s' % filter_kwargs.get('pk'
⚠ 237
⚠ 238          if not results:
⚠ 239              raise NotFound
⚠ 240
⚠ 241          obj = results.pop()
  242
  243          # May raise a permission denied
  244          self.check_object_permissions(self.request, obj)
  245
  246          return obj
  247
  248      def retrieve(self, request, *args, **kwargs):
  249          instance = self.get_object()
  250          serializer = self.get_serializer(instance)
  251          return Response(serializer.data)
  252
  253
```

Figure 2.4: SQL injection in a Python web application presented as a locate exercise, the second of three different challenge types on the SCW platform.



```
    API endpoint for viewing public courses.
    """
    queryset = Course.objects.filter(is_active=True).order_by('start_date')
    serializer_class = CoursePublicSerializer
    permission_classes = (AllowAny, )

    def get_object(self):
        """
        Returns the object the view is displaying.
        """
        queryset = self.filter_queryset(self.get_queryset())

        # Perform the lookup filtering.
        lookup_url_kwarg = self.lookup_url_kwarg or self.lookup_field

        assert lookup_url_kwarg in self.kwargs, (
            'Expected view %s to be called with a URL keyword argument '
            'named "%s". Fix your URL conf, or set the `.lookup_field` '
            'attribute on the view correctly.' %
            (self.__class__.__name__, lookup_url_kwarg)
        )

        filter_kwargs = {self.lookup_field: self.kwargs[lookup_url_kwarg]}
        ~~results = list(queryset.raw(~~
            ~~'SELECT * FROM grades_course WHERE id=%s' % filter_kwargs.get('pk')))~~

        ~~if not results:~~
            ~~raise NotFound~~

        ~~obj = results.pop()~~

        # May raise a permission denied
        self.check_object_permissions(self.request, obj)

        return obj

    def retrieve(self, request, *args, **kwargs):
        instance = self.get_object()
        serializer = self.get_serializer(instance)
        return Response(serializer.data)
```

```
urse.objects.filter(is_active=True).order_by('start_date')
ass = CoursePublicSerializer
asses = (AllowAny, )

t(self):
        """
        Returns the object the view is displaying.
        """
        queryset = self.filter_queryset(self.get_queryset())

        # Perform the lookup filtering.
        lookup_url_kwarg = self.lookup_url_kwarg or self.lookup_field

        assert lookup_url_kwarg in self.kwargs, (
            'Expected view %s to be called with a URL keyword argument '
            'named "%s". Fix your URL conf, or set the `.lookup_field` '
            'attribute on the view correctly.' %
            (self.__class__.__name__, lookup_url_kwarg)
        )

        filter_kwargs = {self.lookup_field: self.kwargs[lookup_url_kwarg]}
        obj = generics.get_object_or_404(queryset, **filter_kwargs)

        # May raise a permission denied
        self.check_object_permissions(self.request, obj)

        return obj

    def retrieve(self, request, *args, **kwargs):
        instance = self.get_object()
        serializer = self.get_serializer(instance)
        return Response(serializer.data)

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    """
    API endpoint for viewing courses.
    """
    serializer_class = CoursePublicSerializer
    permission_classes = (IsAuthenticated, )
```

Figure 2.5: SQL injection in a Python web application presented as a fix exercise, the third of three different challenge types on the SCW platform.

2.5 Course material

The SCW training portal provides training in more than 50 languages and frameworks[2], ranging from Cobol to Go, including languages for web, mobile, cloud, and embedded software. The training content covers 184 different vulnerability types, including those in widely-used lists such as the OWASP Top 10[3], OWASP Top 10 Mobile, OWASP Top 10 Application Programming Interface (API) Security and the Common Weakness Enumeration (CWE) Top 25[4]. The full list of vulnerabilities can be found on the SCW website[5]. This coverage is not homogeneous across all languages. For each language and framework, each relevant vulnerability type is assigned a priority (high, medium, or low). This priority depends on the severity and prevalence of the vulnerability type in this particular language and framework combination.

How well a language is covered then depends on the amount of challenges that cover vulnerability types of different priorities. The minimum requirement for a language to be considered ready for training is three challenges for each category in the OWASP Top 10 categories. A language is considered tournament ready when there are five challenges (two easy, two medium, and one high difficulty) for all vulnerability types with high priority, and two challenges (one easy, one medium) for all vulnerability types with medium priority. There are other requirements still for assessments, specific courses, or the website trial.

For each of the top three frameworks over 450 unique vulnerabilities have been introduced in applications. These frameworks are C# Model View Controller (MVC) (461 vulnerabilities), Java Enterprise Edition (EE) JavaServer Pages (JSP) (475 vulnerabilities), and Java EE Spring (495 vulnerabilities). When multiplied by three (for identify, locate, and fix), there are over 1350 challenges for each of these three frameworks.

2.6 Use in the paved path methodology

The SCW online learning platform is a great educational resource to support the paved path methodology. The platform is *relevant*, the learning context resembles the developer's work context as they are able to receive training in their office or home office and by looking at actual code. The code on the platform is likely to be similar to that of the developer

[2] `https://www.securecodewarrior.com/supported-languages`
[3] `https://owasp.org/www-project-top-ten/`
[4] `https://cwe.mitre.org/top25`
[5] `https://www.securecodewarrior.com/product/supported-vulnerabilities`

due to the wide variety of programming languages, frameworks, and software types that are supported. The exercises teach a developer a secure paved path in their framework of choice. The identify, locate, and fix exercises are all defensive tasks created with the developer in mind.

The platform is also *usable* as there are several features to increase interactivity and engagement, such as the gamified theme, leaderboards, tournaments, achievements, and badges. A structured journey is present in the form of courses, such as the OWASP Top 10 courses. The learning material is presented through multiple choice questions. In newly released exercise, developers are even allowed to discover the answer through trial and error instead of picking from a list of options.

There is certainly enough content available to allow for sufficient repetition so that the concepts can be committed to memory, with some frameworks providing as many as 1350 challenges. However, there is no guidance to find the right balance between repetition and *efficiency*. The exercises often do not match the learning pace of each individual, leading to boredom or frustration. This is apparent from the challenge completion rate, as only 45% of users complete more than 30 challenges, the amount of challenges in an OWASP top 10 course.

When surveyed, some users indicate this possible mismatch in the learning pace. More than 700 respondents were asked to describe their experience using the SCW portal after completing a tournament. To do this, they were able to choose words from a set of options or write their own. Many respondents selected words indicating their engagement such as interactive (55%), engaging (53%), and fun (48%). But some also picked words that could indicate an incorrect learning pace, among which challenging (45%), repetitive (21%), long (7%), and boring (4%). Only 22 (3%) respondents wrote down additional words themselves, some of which indicate mismatches in learning pace. Two users wrote down tedious, two users wrote cumbersome, one wrote frustrating, and one even went as far as to describe their experience as gambling.

In conclusion, the SCW online learning platform is a good educational resource when using a paved path methodology. Its defensive exercises and wide support for different programming languages and frameworks make it *relevant* to the developer's work. The gamification and interactivity keep it *usable* and fun. However, when it comes to the *efficiency* of the training, there is still room for improvement, as currently all users are presented with challenges of the same difficulty regardless of their skill level and learning pace. User feedback indicates that this leads to boredom or frustration for some of the users.

Chapter 3

Intelligent tutoring system

An important aspect of education in the paved path methodology is its efficiency. Educational activities should not keep the developer from their responsibilities for longer than necessary. The efficiency of training on the SCW platform is lacking. Because every individual is presented with the same exercises, they often receive training that is too repetitive or too challenging. I designed an Intelligent Tutoring System (ITS) that recommends exercises to each individual at any point in time to provide them with a more appropriate learning pace. In this chapter, I present the design of the ITS and discuss the used techniques in its implementation in more detail.

If nothing else, take away from this chapter...

I designed an ITS that consists of three algorithmic components, one for exercise selection, and one each for estimation of user ability and exercise difficulty. Exercise selection is achieved through a Collaborative Filtering (CF) algorithm adapted to learning systems. In such an algorithm, a target user's preference for an exercise is predicted based on the preferences of like-minded users. Through the use of the psychometric model of Item Response Theory (IRT), estimation of both user ability and exercise difficulty can be done at once. A sanitized data set of over 9 million solved exercises is used to calibrate these algorithms making up the ITS.

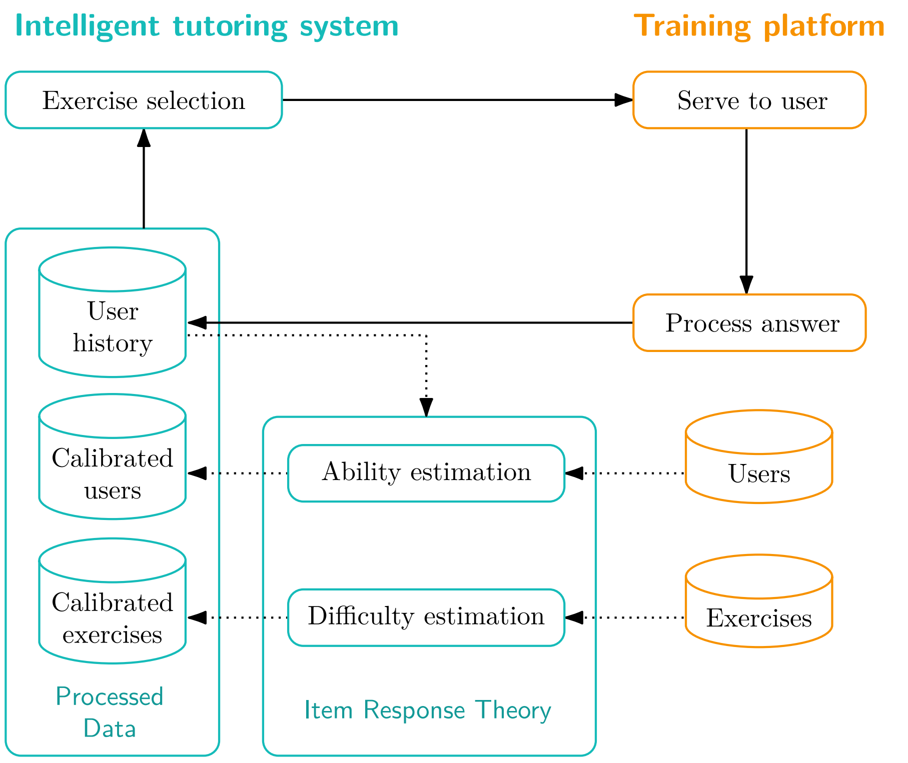


Figure 3.1: The ITS consists of two loops. In the main loop, users are served exercises and their answers are processed, before selecting a new exercise. The user history is then regularly used in a secondary loop to estimate both user abilities and exercise difficulties.

3.1 Design

The design of the ITS, shown in Figure 3.1, extends the existing functionality of the training platform. This existing functionality is depicted in orange on the right side of the figure, while components of the ITS are drawn in blue on the left side. The ITS consists of two loops, it contains three algorithmic components and three types of data collections.

The algorithmic component in the main loop is that of exercise selection. Selecting the optimal exercise is done through an adapted CF algorithm, as will be explained in more detail in Section 3.2. In this technique, a recommendation is derived from historical data of like-minded users. To evaluate this technique, and hence improve it, we need a measure to decide what a good recommendation is.

A useful challenge is a challenge from which the user has learned something and that keeps the user engaged. That is, a good recommendation system should increase the *ability* of the user, and their *engagement*. It is easy to keep track of the engagement of the user. If they continue to play more challenges, that means they stay engaged. However, in order to determine if a recommendation leads to increased ability, we need to be able to continuously measure the ability of each user. Another reason to continuously measure the ability of each user is the temporal aspect to learning. An exercise that is useful to a user at the beginning of their journey is likely no longer an appropriate recommendation once their ability has sufficiently increased. Hence ability estimation is needed to both determine *if* a challenge was useful to a user, and *when* a challenge was useful.

A naive way to achieve an ability estimate is simply looking at the accuracy of each user. A user answering all of the challenges correctly (100% accuracy), is likely to have a higher ability level than a user answering half of them correctly (50% accuracy). If all users completed the exact same challenges, this could give a reasonably accurate representation of their ability level. In fact, that is exactly the reasoning behind Classical Test Theory (CTT) [26]. In a classic test, all examinees are given the same (or equivalent) exercises and their accuracy on the test is an indication of their ability level.

However, on the training platform not all users are completing the exact same challenges and this is not desirable, as that would conflict with the goal of individually tailored recommendations. When users are completing different challenges, accuracy alone is no longer sufficient. It is possible for one user to maintain a high accuracy doing simple challenges, while another user's accuracy is lower but they are completing difficult challenges.

This is also true for exercises, the difficulty of an exercise can not be accurately estimated through the accuracy of users completing it. It is possible for one exercise to have a high accuracy because it is mostly attempted by users of a high ability level, while another is often tried by beginners and hence has a lower accuracy. It is clear that these two remaining algorithmic components in the ITS are tightly coupled. Both are implemented through the use of psychometric models from the field of IRT as explained in Section 3.3. The calibration techniques of IRT use the entire user history and take a while to complete. This is why they are not performed every iteration of the main loop, but at regular intervals in a secondary loop.

3.2 Collaborative filtering

In this section, I discuss the first algorithmic component of the ITS as depicted in Figure 3.1, the component of exercise selection.

There are many possible factors that determine which exercise to select. We can easily imagine some factors that are likely to have a big influence, such as the difficulty of the exercise, the vulnerability type, and the programming language. Research has also shown that individual learning style has an impact on learning performance [27–30]. For other factors, it is more difficult to determine how important they are, or if they matter at all. Some examples are code quality, code legibility, software type, or even just the coding style of the author of the challenge. There are also likely other factors that we are not yet aware of.

For this reason, it is desirable to create a recommendation system that uses a black box approach. With this kind of approach it is not necessary to know which factors determine a good recommendation. There only need to be enough users and challenges, as well as a way to determine if a challenge was useful to a user.

One frequently used technique for recommendation systems is CF. In CF, a target user's affinity for items is used to find other users who are most like-minded. This group's collective affinity for items is then used to predict the target user's affinity for those items.

These types of algorithms are most easily understood through a visual representation. In Figure 3.2, a simple example of a CF algorithm is illustrated. In this figure, the recorded affinity of users i ($i = 0, \dots, 5$) for movies j ($j = A, \dots, J$) is depicted in a two dimensional grid. A green check mark in the grid means that the user enjoyed the movie, a red cross means they did not. There are also many empty spaces as not all users have watched all movies. In order to predict the affinity of a target user $i = 0$ for a target movie $j = B$, the CF algorithm starts by finding the users who are most like-minded, the users who have the most similar recorded affinity. For each user, the algorithm determines for how many movies they have the same affinity as the target user. In Figure 3.2, the affinity of the target user is marked with an orange background, affinity of other users that is the same as the target user is marked with a green background. In the example, user $i = 1$ enjoyed movies $j \in \{A, C\}$. But their affinity for movie C is the only affinity they have in common with the target user. Users $i \in \{2, 3, 5\}$ each have the same affinity as the target user for two of the movies. In this example, they make up the group of people that are most like-minded to the

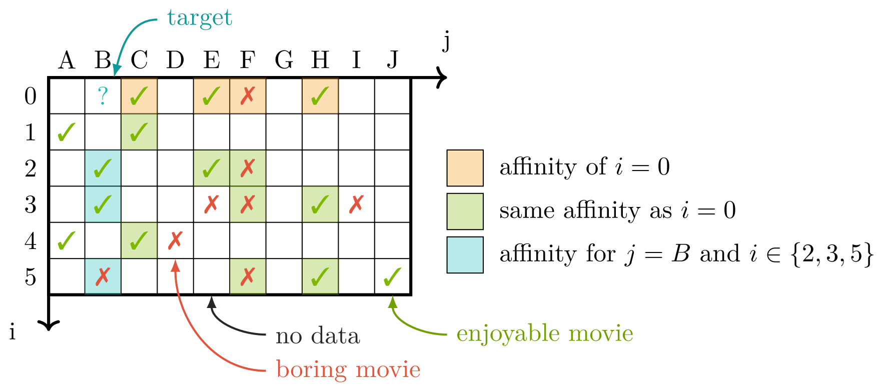


Figure 3.2: Visual representation of the steps to determine if the target movie $j = B$ is a good recommendation for the target user $i = 0$. The CF algorithm first finds all users who have similar preferences for movies as the target user (marked in green). The users who have the most similar preferences are used in a majority vote. In the example users 2, 3, and 5 each had the same preference as the target user for two different movies. The majority of these users enjoyed the target movie (marked in blue), so the algorithm concludes that the target is a good recommendation.

target user. To predict if the target user would enjoy the target movie $j = B$, the algorithm now uses this group's affinity for the target movie, marked in a blue background in Figure 3.2. Two of the most like-minded users enjoyed the movie and one of them did not. Since the majority of like-minded users enjoy the target movie, the algorithm predicts the target user might enjoy it as well.

3.2.1 Adapted to learning systems

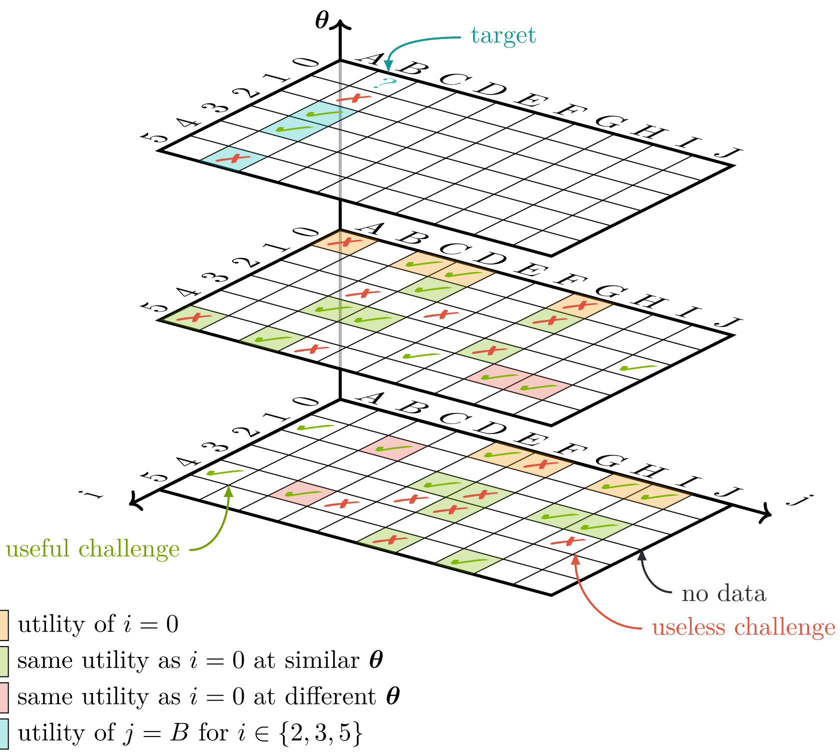
Some adjustments are needed to apply CF to a learning system. In a learning system, a good recommendation is one that allows meaningful learning to take place and at the same time keeps the user engaged. A good recommendation is hence based on the *utility* of a user for an item, rather than their affinity. If the users who are most like-minded increased their ability level through playing this challenge, it is likely a good recommendation.

As mentioned before, learning also has a more apparent temporal aspect to it. An exercise that is useful to a user at the start is no longer an appropriate recommendation once their ability has increased sufficiently. It could hence be beneficial to keep track of the ability level around which a recommendation can be deemed appropriate. The CF algorithm can then only consider users to be like-minded, if they experienced the same utility as the target user within a certain ability range. Users who experienced the same utility for an item but at a sufficiently dissimilar ability level will not be considered like-minded users.

Figure 3.3 illustrates how this adaptation could be achieved on the example algorithm from before. In this figure, the recorded utility that the users i ($i = 0, \ldots, 5$) experienced from challenges j ($j = A, \ldots, J$) is depicted in a two dimensional grid for three sufficiently distinct ability levels $\boldsymbol{\theta}$. A green check mark means that this challenge was useful to the user around that ability level. A red cross means it was not useful, and the user either did not learn anything, or the challenge caused the user to disengage from the training. How the utility of a challenge is determined will be explained in Section 3.4.3. To predict the utility of the target challenge $j = B$ to the target user $i = 0$, the adapted CF algorithm looks for the most like-minded users, the users who experienced the most similar utility.

However, only if the same utility was experienced around the same ability level does it count towards like-mindedness. In Figure 3.3, utility that was experienced by the target user for challenges is marked with an orange background. Similar utility that was experienced around the same ability level is marked with a green background. At the lowest ability level, user $i = 2$ experienced the same utility as the target user for four challenges. User $i = 1$, just like the target user, experienced challenge $j = C$ as useful. However, the challenge was useful to user $i = 1$ at the lowest ability level, while the target user found this challenge useful when their ability level was sufficiently higher. Similar utility like this that was experienced around a different ability level is marked with a red background in the figure. This utility does not count towards like-mindedness. Using this metric for like-mindedness, we find that users $i \in \{2, 3, 5\}$ each experienced the same utility around the same ability level as the target user for four different challenges. They make up the group of users who are most like-minded to the target user.

Beyond this adaptation, the same steps are used to decide the final recommendation. The majority of like-minded users experienced the target challenge as useful around the targeted ability level, marked with a blue background. The algorithm concludes that the target challenge is a good recommendation for the target user.



θ
target
A B C D E F G H I J
0 1 2 3 4 5
?
i
j
useful challenge
no data
useless challenge
🟧 utility of $i = 0$
🟩 same utility as $i = 0$ at similar $\boldsymbol{\theta}$
🟥 same utility as $i = 0$ at different $\boldsymbol{\theta}$
🟦 utility of $j = B$ for $i \in \{2, 3, 5\}$

Figure 3.3: Visual representation of the steps to determine if the target challenge $j = B$ is a good recommendation for the target user $i = 0$. The collaborative filtering algorithm first finds all users who experienced similar utility from challenges as the target user around the same ability level $\boldsymbol{\theta}$ (marked in green). Similar utility at a different ability level is disregarded (marked in red). The users who experienced the most similar utility are used in a majority vote. In the example, users 2, 3, and 5 each experienced the same utility as the target user for four different challenges. The majority of these users experienced the target challenge as useful (marked in blue), so the algorithm concludes that the target is a good recommendation.

3.2.2 Types of collaborative filtering

In the examples until now, the affinity (or utility) of a user for an item was considered binary, the user either liked it, or they did not. In reality, often a more complex scale is used. For example, Netflix movie recommendations use a rating scale between 1 and 5. In this context, the affinity of a user u for an item i is often called the rating r_{ui}. The goal of a CF algorithm is then to make a prediction for this rating $\hat{r}_{ui}$. CF algorithms can be split into two broad categories, memory-based and model-based algorithms, based on how they set out to achieve this goal [31–35].

Memory-based

Memory-based CF algorithms directly use observed ratings to compute predictions. Generally, these algorithms mark a subset of users as neighbours to the target user by calculating the similarity between users [31, 36]. They then use the neighbour's ratings to predict the rating of the target user [35–38].

Memory-based CF algorithms are frequently used in recommender systems and even commercial systems such as the Amazon webstore [32, 33]. The main advantages of memory-based collaborative filtering algorithms are that they are easy to implement, and that new data can be added easily and incrementally. Their biggest shortcoming is a decreased performance for sparse data, when it is difficult to find sufficient neighbours. They also can not make recommendations for new users or items as there is no data to do similarity computations with. In the ITS however, we already need sufficient data to compute the difficulty and ability estimates. In a learning system, the need for an initial calibration phase can not be avoided, so this shortcoming does not impact the design of the ITS.

Five different memory-based CF algorithms are considered in this work. They will be described in more detail in the experiments in Section 4.3. Four of these algorithms are k-nearest neighbours (k-NN) algorithms, they first determine the k nearest neighbours and then use the ratings of these neighbours to compute a prediction. Because these algorithms use an explicit definition of similarity between users, they can be easily adapted to learning systems by changing this definition to take into account the ability of the users. The remaining memory-based algorithm uses the ratings of all users to make a prediction, adapting it to learning systems will be harder, but can still be achieved by processing the data, as will be explained in the description of the experiments.

Model-based

Model-based CF algorithms use statistical and machine learning methods to construct a model, and use this model to make predictions [31, 32, 39, 40]. These models often use techniques to reduce the dimensions of the matrix of user-item ratings. This reduces the scalability and sparsity problems that are experienced by memory-based algorithms [39, 41]. Model-based algorithms are often more accurate, but the construction of the model is often slow and expensive, and they have to be re-built regularly, every time new data is being added incrementally.

Algorithms using clustering techniques are simple examples of model-based algorithms [34, 35, 39, 42]. Users or items are assigned to one or more clusters so that the matrix of user-item ratings becomes a smaller, denser matrix of clusters. It is then possible to use statistics from these clusters to make predictions, for example, by taking the average rating of a cluster. In the experiments of this work, one clustering algorithm is evaluated.

Thanks to their accuracy and scalability, model-based algorithms based on matrix factorization have gained a lot in popularity [38]. These algorithms use the technique of Singular Value Decomposition (SVD) to make a low rank approximation of the original ratings matrix [36, 38, 43]. SVD is a well-established technique in linear algebra and machine learning to identify latent semantic factors. Applying it in the CF domain raises a few difficulties due to the sparsity of the rating matrix, which increases the risk of overfitting.

In the experiments of this work, I evaluated Probabilistic Matrix Factorization (PMF) [44], Non-negative Matrix Factorization (NNMF) [45, 46], SVD [38, 47, 48], and SVD++ [38, 49]. All of these algorithms are explained in more detail in Section 4.3.

3.2.3 Alternative approaches

Many existing alternatives either do not take into account the ability level of the users to make recommendations, or they only take into account the ability level [50].

Adaptive learning systems

Many computerized learning systems already exist, both in commercial offerings and in research literature. Older systems do not consider individual learners needs, but make decisions based on pre-planned instructions for the field of study. As a result, these systems do not provide individual attention to students as a natural (human) teacher would [51]. This inspired the rise of more advanced learning systems that consider both the field and the learner to provide flexibility in the presentation of the educational material. Some such systems have been built to teach computer science concepts, such as debugging [52], cryptographic algorithms [51, 53], programming in C++ [54], or SQL [55].

These systems have varying degrees of intelligence and adaptiveness. In commercial offerings, such as Pluralsight Iris[1], adaptive learning often refers to an initial calibration phase to determine the initial ability level of a user. In other systems, students are allowed to advance to more difficult levels when a sufficiently high accuracy is achieved on exercises in the current level [51, 54]. Even more advanced systems provide adaptive feedback to the users, based on which mistakes have been made in the exercises [52, 53]. Finally, the most advanced systems are able to adapt the difficulty of the exercises more dynamically. Duolingo for example, adapts the difficulty of the of last few exercises in a lesson based on the performance of the student on the previous exercises[2].

All of these systems focus on offering exercises of the appropriate difficulty level, and some also take into account classifications of learning styles [27, 30]. The goal of the ITS, however, is to also pay attention to other potential factors such as the coding style, presentation form, application type, and so on. The discussed systems could potentially be adapted to take into account several of these factors.

Serious games

Several serious games exist for topics related to cybersecurity and social engineering. They usually focus on increasing the awareness of software users of different ages and are not used to train software developers [56–59].

Research in this field mostly focuses on the effect of gamification, and usually adds game elements to static learning content. They usually do not adapt to the users beyond opening up new content after the completion of preceding exercises. While gamification leads to increased engagement, this is not the focus of my research as I believe the SCW platform already has some decent gamification features.

[1] `https://www.pluralsight.com/product/iris`

[2] `https://blog.duolingo.com/`

Content-based recommendation systems

Another type of recommendation system that is related to CF algorithms, is content-based recommenders. These systems analyze item descriptions to identify items that are of particular interest to a user [60]. To do this, they represent both items and users as a vector of characteristics, similar to vectors in the latent space used by model-based CF algorithms. However, in contrast with these CF algorithms, the vectors are not computed in a latent space by the algorithm.

Item characteristics are often already available in the system, or they can be detected through natural language processing. They are often easier to interpret than the dimensions of the latent space in model-based algorithms. On the SCW platform, the framework, language, vulnerability type, and author are examples of characteristics that are readily available.

In order to make a recommendation for a user, items are selected that have similar characteristics to previously liked items by this user. Several algorithms can be used to achieve this, among which nearest neighbour methods and decision trees [60]. In contrast with item-based collaborative filtering, these algorithms compute the similarity between items based on the characteristics of the items themselves. While in CF algorithm the similarity between items is based on the similarity of ratings these items receive by users. This is likely not a good approach for learning systems, where diverse content should be recommended, covering, for example, multiple vulnerability types.

Knowledge-based recommendation systems

Systems that make use of the organisation of learning material are called knowledge-based, or semantics-based recommendation systems. They create a structured knowledge graph or ontology to organise the learning material.

Existing ontologies in software security attempt to organize different security concepts in a broader knowledge graph of computer science. For example, they classify SQL injection as a type of injection attack, SQL security as a type of data integrity, and a Denial of Service (DoS) attack is linked to the availability of the product [61, 62]. While this information can be useful to a developer learning about different security concepts, it can not be used to organise the learning material on the SCW training platform.

Many vulnerability types require a broad knowledge of varying aspects of software development, such as the operating system, communication protocols, language and framework specifics, software architecture, and more. On the SCW platform, however, we assume the users have sufficient knowledge of these domains, and require education in software security only. With this assumption, I do not see a need to create a knowledge graph for software vulnerabilities. It is my belief that most vulnerabilities are unrelated to each other, in the sense that it is possible to understand and master each of them without the need to learn about the other. Current education efforts often focus on the OWASP top 10, in which vulnerabilities are ranked mostly based on their prevalence in practice. This is strong evidence for the fact that the order they are being taught to developers is not of big importance.

Computerized Adaptive Tests

Computerized Adaptive Tests (CATs) are computer-based tests that adapt to the ability of the examinee. They continuously estimate the ability level and serve the next test item based on the current estimate. While there are techniques from CATs that are useful to us, as will be explained in Section 3.3, the item selection is not appropriate for use in the ITS.

This is because the goal of a test is to estimate the ability of an examinee. CATs are able to maintain a higher precision of ability estimation while being about 50% shorter compared to Fixed Item Tests (FITs) [63]. To estimate the examinee's ability in such an efficient way, CATs select the next item in a test based on which one provides the most information about the examinee. These are the items for which the probability of a correct answer is around 50% [64, 65].

This goal is of course different from that of the ITS which is to motivate and engage the users. In fact, the opposite is even true, tests are inherently not very motivating. Certainly that is the case for CATs, where the examinee is only expected to correctly answer half of the test questions. Research has shown that engagement can be improved, and anxiety reduced, by choosing items for which the probability of a correct answer is higher (e.g. 70%) [65]. Still, the item selection algorithm in CATs only takes into account the difficulty of the items. As discussed before, we want the ITS to possibly take into account other aspects of the exercises, such as coding style, author, or application type.

3.3 Difficulty estimation and ability estimation

In the previous section, I discussed the component of exercise selection, the first algorithmic component of the ITS as shown in Figure 3.1. This algorithmic component is implemented through the use of a CF algorithm. In order to use this algorithm effectively in a learning system, we need an accurate ability measure. This ability measure is necessary to both determine *if* a challenge was useful, and *when* a challenge was useful. In this section, I discuss ability estimation, together with difficulty estimation, the two remaining algorithmic components of the ITS. I explain how both can be implemented simultaneously by using IRT, a technique borrowed from the field of psychometrics. This field of study focuses on the objective measurement of skills, knowledge, and abilities, often with the goal to create better computerized tests.

The goal of a test is to estimate the ability of an examinee. With CATs, this can be done with higher precision while using less exercises than classic FITs. To achieve this, CATs continuously adapt the exercises to the estimated ability level of the examinee. An overview of the steps taken by a CAT is shown in Algorithm 1.

Algorithm 1: A computerized adaptive test

```
  input  : calibrated item bank $I$
  output: ability level $\boldsymbol{\theta}$
1 Set $\boldsymbol{\theta}$ to entry level;
2 while termination criterion not met do
3     Select optimal item $i$ from $I$ based on $\boldsymbol{\theta}$;
4     Present $i$ to examinee;
5     Update $\boldsymbol{\theta}$ based on all prior answers;
```

Some key components are needed to create such a test: calibrated test items, a termination criterion, a starting point or entry level, an item selection algorithm, and an ability estimation algorithm. We can easily see parallels between a CAT and the ITS. First, test items in a CAT need to be calibrated, similarly to the exercises in the ITS. Secondly, it is necessary in both systems to continuously estimate the ability of the users. Finally, there is also a selection algorithm that determines which item the user is shown next. However, the item selection algorithm in a CAT is designed with a different goal in mind, as discussed in Section 3.2.3.

CATs frequently use the psychometric model IRT. This model not only allows calibration of both users and items, but because they are placed on the same scale, the results can easily be used for item selection as well. Although it will not be used for exercise selection in the ITS, the model is certainly useful for accurately estimating exercise difficulty and user ability.

3.3.1 Item response theory

IRT is a model for measuring psychological *latent traits*, i.e. unobservable characteristics such as ability or competence level. The model estimates these latent traits by means of *manifest* (observable) variables and statistical psychometric models. This is done based on the mathematical relationship between the latent traits and the manifest variables. A user with a higher ability level (latent trait) is more likely to answer more questions, and more difficult questions, correctly (manifest variables). This relationship can be written down as a mathematical function, called the Item Response Function (IRF). It describes the possibility of observing each possible answer as a function of person ability levels and exercise parameters.

$$P_{jk}(\boldsymbol{\theta}_i, \boldsymbol{p}_j) = Pr(X_{ij} = k | \boldsymbol{\theta}_i, \boldsymbol{p}_j) = f(k, \boldsymbol{\theta}_i, \boldsymbol{p}_j) \tag{3.1}$$

In Equation 3.1 the ability level of a person i ($i = 1, \dots, I$) is represented by a multivariate vector of latent traits $\boldsymbol{\theta}_i$. $\boldsymbol{p}_j$ is the set of parameters of exercise j ($j = 1, \dots, J$). X_{ij} is the answer of person i for exercise j, with k representing one possible answer. For dichotomously scored exercises, meaning there are only two possible answers (e.g. true/false), $k \in \{0, 1\}$. Exercises with more than two options are called polytomously scored exercises (e.g. multiple choice). For these exercises $k \in \{0, \dots, K_j\}$. There are many mathematical functions that can be used to describe the IRF, each resulting in a different IRT model. The most common is called the Rasch model, an extension of this model, called the two-parameter logistic model (2PL) is used in the ITS. These models will be explained in the next section.

The Rasch model and 2PL model are dichotomous models, which means the items are scored as only having two possible outcomes, correct or incorrect. It is necessary to use a dichotomous model in the ITS because some important data is missing to use polytomous models effectively. Polytomous models are most effective when the number of possible options for the multiple choice question is limited to three or four. The incorrect options need to be deliberate and able to mislead a person with a lower ability level.

On the SCW training platform, this is not the case as will be explained in Section 3.4. Because of this lack of useful polytomous information, I decided to use the dichotomous 2PL model. For dichotomous models, there are only two outcomes, $k \in \{0, 1\}$, representing correct and incorrect. That means the IRF in Equation 3.1 can be reduced to

$$P_{j1}(\boldsymbol{\theta}_i, \boldsymbol{p}_j) = Pr(X_{ij} = 1|\boldsymbol{\theta}_i, \boldsymbol{p}_j) = P_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j), \tag{3.2a}$$

$$P_{j0}(\boldsymbol{\theta}_i, \boldsymbol{p}_j) = Pr(X_{ij} = 0|\boldsymbol{\theta}_i, \boldsymbol{p}_j) = 1 - P_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j) = Q_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j), \tag{3.2b}$$

such that P_j represents the probability of a correct response for exercise j, and Q_j the probability of an incorrect response.

3.3.2 Rasch model

Several mathematical functions can be used to characterize the IRFs in Equations 3.2a and 3.2b. The most commonly used are logistic distribution functions, resulting in the so-called Rasch model [66].

In its simplest form the Rasch model takes only one parameter $\boldsymbol{p}_j = b_j$ representing the difficulty of the exercise. The IRF using a one-parameter logistic distribution function is shown in Equation 3.3.

$$P_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j) = Pr(X_{ij} = 1|\boldsymbol{\theta}_i, b_j) = \frac{\exp(\theta_i - b_j)}{1 + \exp(\theta_i - b_j)} \tag{3.3}$$

The resulting model is called the one-parameter logistic model (1PL). In Figure 3.4 the IRF of the 1PL model is plotted for three exercises with various values for the difficulty parameter. In this model, for a fixed ability level $\boldsymbol{\theta}$, the probability of a correct answer P_j is lower for exercises with a larger difficulty b_j. The probability takes the value of 0.5 when the ability level of the user exactly matches the difficulty of the item. This is a result of placing the user abilities and the item difficulties on the same scale.

There are more characteristics to an exercise than its difficulty. One characteristic that is useful to help estimate the ability of users is the discriminative ability of the exercise. This characteristic represents how good an exercise is at differentiating between users of varying ability levels. An extension to the 1PL model exists that includes this parameter, resulting in the 2PL. This second parameter a_j of an item is a multiplicative parameter, as shown in the IRF in Equation 3.4.

$$P_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j) = Pr(X_{ij} = 1|\boldsymbol{\theta}_i, a_j, b_j) = \frac{\exp\left[a_j(\theta_i - b_j)\right]}{1 + \exp\left[a_j(\theta_i - b_j)\right]} \tag{3.4}$$

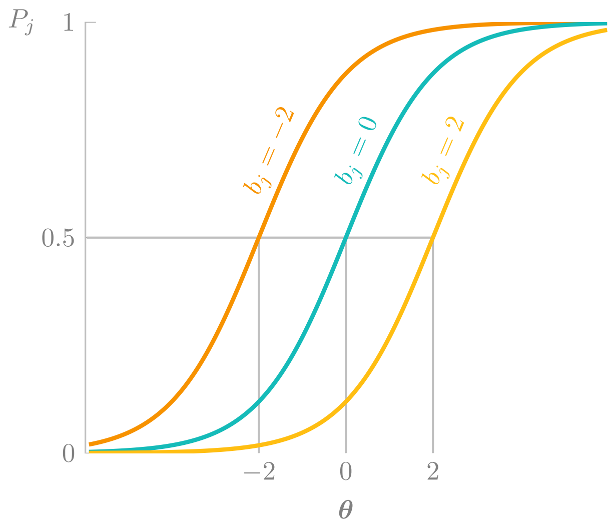


Figure 3.4: Three IRFs in the 1PL model, with different difficulty parameters b_j. The difficulty parameter represents the difficulty of the exercise. For a fixed ability level, e.g. $\boldsymbol{\theta} = 0$, a higher difficulty means a lower probability P_j of getting a correct answer. The probability of a correct answer is 50% when $b_j = \boldsymbol{\theta}$.

In Figure 3.5 the IRFs are plotted for three exercises with the same difficulty $b_j = 0$ but various discriminative abilities. For all three IRFs the probability of a correct answer is still 0.5 for users with an ability level equal to the exercise difficulty. However, the second parameter influences the steepness of the curve. For larger discrimination parameters such as $a_j = 2$, a smaller increase in ability $\boldsymbol{\theta}$ around the exercise difficulty level $b_j = 0$ leads to a more notable increase in probability of a correct answer P_j. Such an exercise discriminates better between low and high ability users. On the other hand, exercises with lower discrimination values, like $a_j = 0.5$ result in flatter IRFs and do not allow as easily for such discrimination.

Model calibration

Both the 1PL and 2PL models hold parameters of two types: item parameters $\boldsymbol{p}_j$ and person parameters $\boldsymbol{\theta}_i$. With enough data, both sets of parameters can be accurately estimated. First, the item parameters are estimated independently of the ability levels, this is called the *model calibration*. Next, the ability levels are estimated while keeping the item

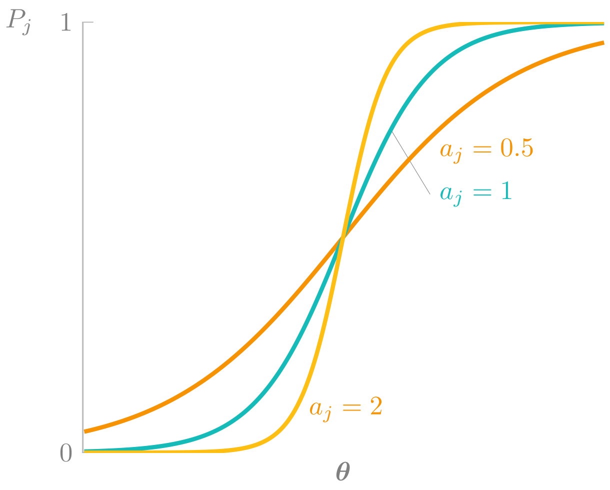


Figure 3.5: Three IRFs in the 2PL model, with equal difficulty parameter b_j but different discrimination parameters a_j. The discrimination parameter represents how well an exercise can differentiate between users of different ability levels $\boldsymbol{\theta}$. A higher value means a steeper increase in probability of a correct answer P_j around the difficulty level $\boldsymbol{\theta} = b_j$.

parameters fixed.

IRT offers several model calibration techniques, mostly designed for item banks with several dozens of items. The larger size of the item bank of SCW will cause longer execution times and more difficult convergence to stable estimates. I solve this problem by splitting the item bank to several smaller item banks, for example one for each framework on the platform. The resulting consequences are discussed in Chapter 4.

The calibration of the model consists of tuning the model parameters to maximize the likelihood of the observed data. More formally, model calibration is maximizing the likelihood of the model $L(\boldsymbol{\theta}, \boldsymbol{p})$ with respect to all item parameters $\boldsymbol{p} = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_J)$. Because this likelihood is also dependent on all person parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_I)$, direct maximization is not possible. There are several possible techniques that can be used to overcome this. The most well-known are Joint Maximum Likelihood (JML), Conditional Maximum Likelihood (CML), and Marginal Maximum Likelihood (MML) [64].

The JML algorithm iteratively maximizes the full likelihood with respect to both person and item parameters until convergence is reached.

CML relies on properties specific to the Rasch model to replace unknown ability levels with known sufficient statistics which then allows for the estimation of the item parameters without requiring the estimation of the person parameters.

MML is formulated under the assumption that the ability is a random parameter. In contrast to the CML, it can be used for other IRT models besides the Rasch model [67]. It does not replace the person parameters by sufficient statistics, but aims to integrate out the person parameters from the maximization process [64, 67]. A prior distribution for the ability parameters $f(\boldsymbol{\theta})$ is used to compute the marginal likelihood (or the expectation) of a response pattern as shown in Equation 3.5.

$$P_j(\boldsymbol{p}_j) = \int_{\mathbb{R}} P_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j) f(\boldsymbol{\theta}_i) d\boldsymbol{\theta}_i \tag{3.5}$$

The prior distributions are often chosen as normal distributions. The item parameters can then be estimated by maximizing the full likelihood, calculated as the product of all marginal pattern likelihoods. Computing these marginal response pattern distributions was conceptually complex until efficient Expectation-Maximization (EM) algorithms were implemented [64].

In this work, I used the MML approach as it has many advantages, such as its applicability to many types of IRT models, its ability to compare the fit of different models, and its ability to handle perfect response patterns (all correct or all incorrect answers).

Ability estimation

Once the item parameters are calibrated, they are set as fixed. Estimating the person parameters is done through maximizing the *likelihood function* shown in Equation 3.6, with respect to θ.

$$L(\theta) = \prod_{j=1}^{J} \prod_{k=0}^{K_j} P_{jk}(\theta, \boldsymbol{p}_j)^{Y_{jk}} \tag{3.6}$$

In this equation, Y_{jk} equals one if the response category k was chosen for item j and zero otherwise.

Note that since the item parameters $\boldsymbol{p}_j$ are already calibrated and now set as fixed, the function only depends on the person parameters θ. Maximizing this function is equivalent to maximizing its logarithm, the so-called *log-likelihood function*, as shown in Equation 3.7.

$$l(\theta) = \sum_{j=1}^{J} \sum_{k=0}^{K_j} Y_{jk} \log P_{jk}(\theta, \boldsymbol{p}_j) \tag{3.7}$$

For dichotomous items ($k \in \{0, 1\}$), such as in our case, this can be written as:

$$l(\theta) = \sum_{j=1}^{J} Y_{j1} \log P_j(\theta, \boldsymbol{p}_j) + Y_{j0} \log Q_j(\theta, \boldsymbol{p}_j) \tag{3.8}$$

There are several calibration methods to maximize this likelihood, describing them is out of the scope of this book. The method applied in this work is the Weighted Likelihood Estimator (WLE) [64].

3.3.3 Alternative approaches

IRT is often used to adapt the learning material to the ability level of the student in e-learning systems [50, 68]. Often the IRT ability level is used to determine the appropriate difficulty level of the exercises that should be presented to the user [50, 68, 69]. In another approach, the IRT ability level was used as a weight in CF algorithms so that that the users with greater ability level have greater weight in the calculation of the recommendations than the users with less knowledge. This approach assumes that users of greater ability level, such as teachers, are more able to assess the utility of an item than users of lower ability level. In this work, however, users do not rate the items explicitly but instead the utility is determined from learning outcome and engagement, as will be explained in Section 3.4.3.

Despite the popularity of IRT, some alternative approaches exist to determine the ability level of a user.

Classical Test Theory

In CTT, all users have to complete the same exercises and their accuracy on the questions is used as an indicator for the ability level. Classic test theory assumes that each person has an unobservable true ability score T that would be the result of a test if there are no errors in the measurement. In a test, the observed score X is measured instead. This score is defined as the sum of the true score and a measurement error E. According to CTT, the true score is impossible to obtain, but several techniques exist to estimate the reliability of a test.

Besides the lower accuracy of ability estimation, CTT is also not appropriate to use in the ITS because it requires fixed item selection for all the users. By doing this, some users are guaranteed to feel bored or frustrated since it is impossible to select items of the appropriate difficulty level for all of the users simultaneously.

Other IRT models

Three- and four-parameter logistic models In this work we are using the 2PL model of IRT. However, further extensions exist on the 2PL model that add additional parameters. The three-parameter logistic model (3PL) and four-parameter logistic model (4PL) add parameters that influence the lower and upper asymptotic behaviour of the IRF [64]. The lower asymptote parameter c_j is sometimes called the pseudo-guessing parameter. It allows the probability of a correct answer for infinitely small ability levels to be a positive probability instead of zero in the 1PL and 2PL model. Even users of extremely low ability level still have a non-zero probability of answering the exercise correctly. In reality, this can of course be achieved through guessing the correct answer in case of multiple-choice.

On the opposite side of the curve, the upper asymptote parameter d_j allows the maximal probability to be lower than one. This parameter is called the inattention parameter. The 4PL model allows users of extremely high ability level to have a probability of less than 1 of answering the exercise correctly. This can be explained as the user being inattentive or hurried when answering the question.

The resulting IRFs are shown in Equations 3.9 and 3.10.

$$\begin{aligned} P_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j) &= Pr(X_{ij} = 1 | \boldsymbol{\theta}_i, a_j, b_j, c_j) \\ &= c_j + (1 - c_j) \frac{\exp\left[a_j(\theta_i - b_j)\right]}{1 + \exp\left[a_j(\theta_i - b_j)\right]} \end{aligned} \tag{3.9}$$

$$\begin{aligned} P_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j) &= Pr(X_{ij} = 1 | \boldsymbol{\theta}_i, a_j, b_j, c_j, d_j) \\ &= c_j + (d_j - c_j) \frac{\exp\left[a_j(\theta_i - b_j)\right]}{1 + \exp\left[a_j(\theta_i - b_j)\right]} \end{aligned} \tag{3.10}$$

IRFs of the 3PL model with varying pseudo-guessing parameters are plotted in Figure 3.6. In Figure 3.7, three IRFs are plotted with varying values for the inattention parameter. The addition of these parameters is likely to improve the fit of the model to the data. However, the pseudo-guessing and inattention parameters of exercises are not of much use

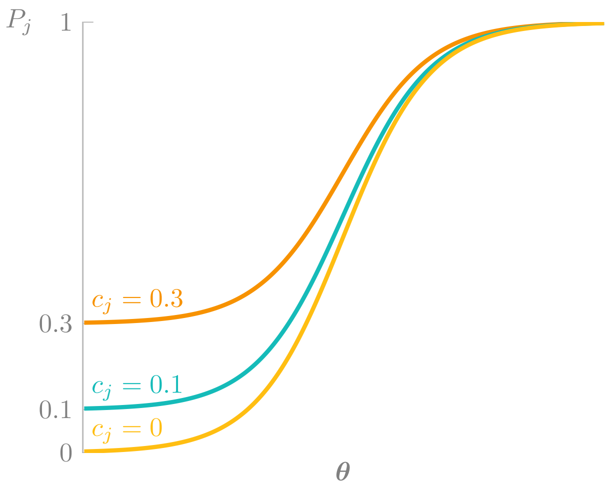


Figure 3.6: Three IRFs in the 3PL model, with equal difficulty and discrimination parameters but different pseudo-guessing parameters c_j. The pseudo-guessing parameter represents the probability that a user is able to guess the correct answer to a question. A non-zero pseudo-guessing parameter means a non-zero probability of a correct answer P_j, even for users of extremely low ability level $\boldsymbol{\theta}$.

in the ITS or the SCW portal in general, and hence the 2PL model is chosen.

Polytomous IRT models In the 2PL model, we are only concerned with whether or not the selected answer is correct. However, even if an incorrect option is selected, it is often possible to use this information to refine the estimate of the ability level of the user based on which incorrect option was chosen [64]. This is the intent of polytomous IRT models. There are many polytomous models available but they generally require scoring in a way that partial credit can be given for incorrect answers [70]. This makes sense if one considers scoring essays based on quality, or giving partial credit in mathematical questions for completing some of the steps.

On the SCW portal this could be achieved by ranking the options in the multiple choice questions from the most incorrect to the most correct. However, currently this is not the case, so we have no indication of which incorrect answer is closest to correct. Furthermore, historically the incorrect answers have not been logged in the data collection, only the amount of incorrect guesses before a correct answer is given. In case

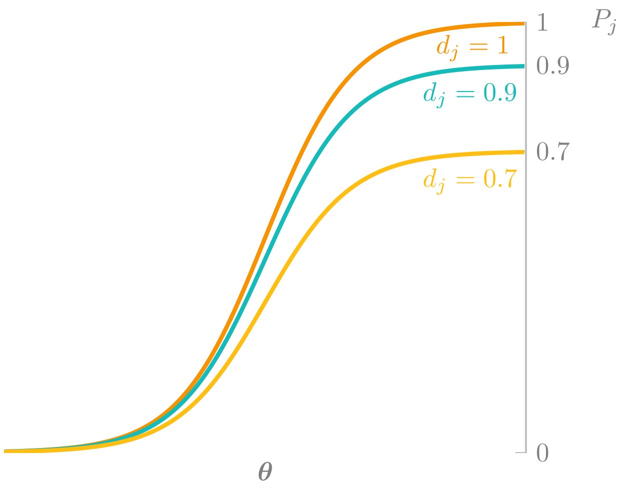


Figure 3.7: Three IRFs in the 4PL model, with equal difficulty, discrimination, and pseudo-guessing parameters but different inattention parameters d_j. The inattention parameter represents the probability that a user answers incorrectly due to inattention. An inattention parameter different from 1 means that the probability of a correct answer P_j is never 1, not even for users of extremely high ability level $\boldsymbol{\theta}$.

the user gave up before finding the correct answer (or in assessments where only one guess is available), the last guess is logged. Polytomous models could result in more accurate results, but currently they cannot be used with the available data.

Multidimensional IRT models While the ability level has been introduced as a multivariate vector of latent traits $\boldsymbol{\theta}_i$, in Section 3.3.1, in the current implementation this has been implemented as a scalar. In reality, the ability of a user regarding software security is not accurately represented as a single value. When the ability estimate is represented as a vector, a value can be obtained for each dimensions of the skill. For example, the ability of a user for each of the vulnerability categories can be represented as a scalar. Using a multidimensional representation for the ability level like this, is not only likely to result in more accurate estimates, but also allows more granular decision making of the appropriate difficulty for an item in the ITS. However, sufficient data needs to be available to have an accurate measurement for each of the dimensions which would limit the amount of users that can be accurately assessed. I have opted to keep as much data as possible to train the CF algorithm instead.

Response-time IRT models Response-time IRT models also take into account how much time each user takes to answer a correct answer. However, as mentioned before, there have been several bugs present in the time tracking features on the platform and this data is currently unreliable.

Furthermore, time pressure varies in different modes on the platform. While users can generally take as much time as they prefer when answering questions in training and assessments, in tournaments there is a limited time to complete the questions.

Elo system

The Elo rating system is a method for calculating relative skill levels between players of a zero-sum game [71]. It is named after its inventor Arpad Elo. The system is widely used in sports, games, and videogames, such as chess, American football, basketball, Major League Baseball, table tennis, Scrabble, Counter Strike: Global Offensive, and League of Legends.

Similarly to IRT, the ability level is not measured directly, but it is inferred from wins, losses, and draws against other players or teams. Based on the current ability levels, the expected outcome of a match-up is predicted. When the actual outcome differs from this expectation, the ability level is updated [72]. By how much it is updated depends on the difference between the ability levels, and in some cases by the observed skill difference. For an overwhelming victory a bigger increase in ability will often be awarded than for a near win.

The Elo system is designed for symmetric match-ups such as player versus player, or team versus team. A zero-sum game is a mathematical representation of a situation in which an advantage that is won by one player is lost by the other. While IRT sets both users and items on the same scale, the training platform can not be considered a zero-sum game. One challenge is played by many more players than a single player usually plays challenges.

Although adaptations exist for asymmetric games [73], I expect the ability estimates to converge slower and the difficulty estimates of items to be less stable than with IRT. IRT takes into account the entire response pattern, hence it is able to estimate the outcome of a challenge again later, based on new information. The Elo system only takes into account the current ability of the user, which might still be inaccurate at the time of playing. As a result, potentially large updates are made to the difficulty of the items that are presented to this user.

3.4 Data

The data used to create the ITS is extracted from the developer log of the SCW training platform.

While this database was not originally intended to be used for analytics, and other data sources have been set up since, lots of useful data is present in this database and it is by far the largest collection.

Each challenge on the platform has a unique identifier (id). This id is tied to the vulnerability in a code fragment, not to the way it is presented to the user (L1-L5). For each challenge the language and framework are known, as well as the category and subcategory of the vulnerability. The difficulty assigned to this challenge is also known but it is not an accurate representation. The way this difficulty is determined is explained in more detail in Appendix A.

The multiple choice options for identify challenges are randomly generated each time they are presented to a user on the website. For the fix challenges the multiple-choice options are fixed, and determined by the author of the challenge. There is no clear order to the multiple-choice options. There is only one correct option, and all others are incorrect, with no distinction between which incorrect option is the closest to the correct one, or the most misleading option.

Each user has a unique id that is a hashed, this way it is not possible to identify the person behind this user id in compliance with the General Data Protection Regulation (GDPR). Some information on the user is available, but it is not used in the design of the ITS. For example, the time zone and the chosen country as the *home base* in the gamification features. For each user it is also possible to look up information about the company they are working for, such as the size of the company and its main industry.

3.4.1 Data collection

Whenever a user tries to solve a challenge, a number of variables is written to the developer log. For each challenge attempt, the unique user id is logged, as well as the challenge id and the way this challenge was presented to the user (L1-L5). Each play mode (tournament, training, assessment) is logged in a separate collection. A number of performance metrics are logged as well, such as the outcome (correct or incorrect), the amount of attempts needed, and the amount of hints used.

The time spent to complete the challenge is logged as well. However upon inspecting this, there seem to have been several bugs present in the past, making this metric not usable. Furthermore, even for the more recent data where these bugs have been resolved, there is no way to find out whether the user was actively trying to solve the challenge during this time, or doing something else. It is also not possible to know how much of this time was spent reading the hints.

More than 12 million challenge attempts have been written to the developer log.

3.4.2 Data pre-processing

This vast collection of data includes users who only completed a small number of challenges, such as during the free trial. For these users, we cannot accurately predict their ability level, nor can we learn much about their preferences. Both the CF algorithm and the 2PL model are only successful when the user has a sufficiently long history on the platform. For this reason, we are only using challenge attempts done by users who completed at least 20 challenges, a recommended minimum length to achieve reasonable accuracy for the 2PL model. Users who have completed at least 20 challenges will be called active users from here on. Out of the 175,979 unique users who have completed at least one challenge on the platform, 95,591 are considered active users whose behaviour on the platform will be used to create the ITS.

A similar argument can be made for the challenges. To accurately calibrate the difficulty of a challenge enough users of varying ability levels need to complete it. The ability level of these users needs to be known. To calibrate the difficulty of an exercise, IRT recommends a minimum of 500 responses to an item. As a result we are filtering out all challenges that are not completed by at least 500 active users. Out of the 19,782 exercises 9,144 remain, 40 of the 50 frameworks are still represented in this data set. On the SCW training platform, there are multiple modes, each with their own rules. In training and tournament, hints are available and multiple attempts are allowed, while in assessments there is only one attempt possible. The 2PL model, on the other hand, expects dichotomous responses. To map challenge attempts of all modes fairly to this binary outcome, any challenge attempt is only considered correct if it was answered correctly without use of any hints on the first attempt. In the rest of this book this will be implied when it is written that a challenge is answered correctly.

3.4.3 Data annotation

The exercise difficulties and user ability levels as estimated by the 2PL model can be used to determine the utility of an item for a user. A scale of 1 to 5 was chosen, and each item is assigned a neutral utility of 3 by default. This utility is updated based on the performance of the user on this item, and on related items later in time.

Flow The first influence on the utility of an item is user engagement. If the item is considered likely to keep the user in flow, its utility is increased. If the item is likely to make users feel frustrated or bored, its utility is decreased. Whether or not an item is likely to keep a user in flow is calculated using the probability of a correct answer. Before each item is played, the difficulty of this item and the current ability level of the user are used to determine this probability through the IRF. The utility is then updated based on this probability as well as the actual outcome of the answer. If the probability of a correct answer is lower than 50%, and the user did indeed answer the item incorrectly, this item's utility is lowered for risk of frustration. If, on the other hand, the probability of a correct answer is larger than 80% and the item is answered correctly, this item's utility is lowered for risk of boredom. These values are in line with research on the engagement of examinees in CATs [65, 74, 75]. The utility in both these cases is decreased by one.

This decrease in utility is immediately nullified if the next item is played in short succession. When the next item is played within 40 minutes, the upper scale of a reasonable play time for a challenge, this is evidence of continued engagement by the user. In that case, the utility of the previous item is increased by one as a reward.

Increased ability If a user learns something new from an item, this should also lead to increased utility. However, with IRT, and any other approximation methods, the estimate of the ability is never increased based on an incorrect answer. The only reason to increase the ability estimate, is when there is evidence of this increase, i.e. when the user answers a difficult item correctly. In reality, however, learning takes place before this correct answer, and the actual (unobservable) ability level has increased earlier in time, so that the user was able to answer this difficult item correctly. In line with this observation, items are updated retrospectively based on answers to related items later in time.

If an item is answered correctly, the utility of the previous item in the same vulnerability category is increased based on the difference in difficulty. Similarly, the utility of the previous item in the same category is decreased for an incorrect answer.

Putting the rewards and punishments for flow and increased ability together results in the final algorithm to determine the utility of an item as shown in Algorithm 2. The `min` and `max` functions are added to ensure utility ratings stay within the 1 - 5 scale.

Algorithm 2: Utility of challenges

```
input  : current item $i$
         probability of correct answer $p$
output: utility $u_j$, for items $j \leq i$
1 $u_i \leftarrow 3$
2 $u_j \leftarrow$ previous item about same vulnerability category
3 if $i - 1$ played less than 40 minutes ago then $u_{i-1} = u_{i-1} + 1$;
4 if $i$ answered correctly then
5     if $p > 0.8$ then $u_i = u_i - 1$;
6     if $d_i > d_j$ then $u_j = \min(5, u_j + (d_i - d_j))$;
7 else
8     if $p < 0.5$ then $u_i = u_1 - 1$;
9     if $d_i < d_j$ then $u_j = \max(1, u_j + (d_i - d_j))$;
```

Annotating the items like this results in a reasonable distribution of the ratings. A large portion of the items is rated around the middle rating of 3, but a significant amount of items also end up with a lower or higher rating. A clear bias exists for some users, for whom the ratings are either consistently higher or consistently lower. This can be explained by the (mis)match between these users' ability levels and the current item selection of the default courses or tournaments. Some users have an ability level that is significantly higher, and they are consistently shown items that can be considered too easy for them, resulting in consistently lower ratings. For other users, the current selection happens to be about right for their skill level, and hence their ratings are consistently somewhat higher.

Chapter 4

Experiments

Several techniques and algorithms have been combined in the design of the ITS. With the implementation and evaluation of each of these, new insights were gained into the mental model of the developer. In this section, I describe the four experiments I conducted to evaluate and fine-tune each component in the ITS.

If nothing else, take away from this chapter...

In the first experiment, I evaluated the results of the 2PL model through statistical analyses. I found that the assigned difficulty of an exercise on the SCW platform does not accurately represent the actual difficulty as experienced by the users. The statistical tests show that the actual difficulty does depend on the framework of the exercise, the category of the vulnerability, and the presentation of the exercise to the user.
The second experiment showed that out of the tested approximation methods, the one invented in this work is the best to achieve a fast ability estimate at each point in time. The mean error stays below 10% even after more than 200 completed challenges.
In the final two experiments, I tested the performance of different CF algorithms and their adaptations to learning systems. The adaptations caused an increase in prediction accuracy between 3.9% and 13.7%, depending on the algorithm. The best performing algorithm in the end is the k-NN baseline algorithm, achieving a mean absolute error of 0.4206.

4.1 Rash model

4.1.1 Goals and research questions

The main *goal* of the first experiment is to discover correlations between the difficulty of an exercise as estimated by the 2PL model, and characteristics of that exercise. The *purpose* is to gain insights in the mental model of the developer, and discover which languages, frameworks, or vulnerability types are typically more difficult. The results of this experiment can be used in the ITS to approximate the difficulty of an exercise when there is no sufficient data available to obtain an estimate with IRT.

The above goal can be achieved by means of an experiment aimed at answering the following questions: Is there a statistically significant correlation between the difficulty of a challenge and its

- **Q1** assigned difficulty on the SCW platform?
- **Q2** vulnerability category?
- **Q3** vulnerability type (subcategory)?
- **Q4** language and framework?
- **Q5** presentation form (locate, identify, fix)?

It is expected that the last four characteristics have a significant influence on the difficulty of a challenge. Some vulnerabilities are more difficult to detect, understand, or fix than others. This is evident, for example, from the OWASP Top 10 pages, where each category is assigned a score for exploitability, prevalance, detectability, and technical impact. There is no clear explanation as to how these scores are determined. Personal and anecdotal evidence also shows that it is more difficult to get security right while working in certain languages and frameworks.

The first variable, the assigned difficulty on the SCW platform, is expected to have little impact on the actual difficulty. As explained in Appendix A, this difficulty only represents the probability of a correct answer in case of a blind guess. It does not take into account the contents of the exercise, only the number of options to choose from. While the amount of options is expected to have some influence on the actual difficulty, this effect is likely to be small compared to other characteristics of the exercise.

4.1.2 Experimental set-up

In Section 3.3, it was previously mentioned that calibration techniques for the 2PL model are mostly designed for item banks with several dozens of items. The larger size of the item bank of SCW will cause longer execution times and more difficult convergence to stable estimates. I solved this problem by splitting the item bank into several smaller item banks and calibrating each of them separately.

The downside to this approach is that the 2PL model does not use an absolute scale. This is evident from the equation for the 1PL model in Equation 3.3, repeated here for convenience in Equation 4.1.

$$P_j(\boldsymbol{\theta}_i, \boldsymbol{p}_j) = Pr(X_{ij} = 1|\boldsymbol{\theta}_i, b_j) = \frac{\exp(\theta_i - b_j)}{1 + \exp(\theta_i - b_j)} \tag{4.1}$$

Adding or subtracting the same value from both the difficulty parameter b_j and the ability level θ_i will cancel out and result in the same probability of a correct answer P_j. In other words, the origin of the scale can be chosen arbitrarily. In practice, the origin is often set to the mean of the ability estimates [64]. I used the R package `TAM` to estimate the item and person parameters [76]. In the documentation of this package there is no information about the chosen scale. Based on the implementation that is available on GitHub[1], the scale is set to the mean of the difficulty estimates after the first iteration of the calibration process. For each item bank that is calibrated separately, the results can hence not be easily compared to other item banks, as they each have their own scale.

However, for this experiment it is necessary to compare the difficulty of challenges across the entire platform. To be able to do this, the entire item bank has to be calibrated at once. The necessary computations took over 73 hours to complete, confirming once more that it would not be feasible to use in the ITS.

Statistical significance

With this single 2PL model calibrated, there is a difficulty estimate for every challenge on the platform. It is now possible to group these challenges according to different characteristics and compute the mean difficulty of each group. Before we can compare the mean difficulties for each characteristic, however, the results have to be tested for statistical significance.

[1] `https://github.com/cran/TAM/blob/master/R/tam.mml.2pl.R`

Which test can be used to do this, depends on the values each characteristic can take. The difficulty as assigned on the SCW platform takes on values between 0 and 100, and is hence a numerical variable. The vulnerability category, vulnerability type, framework, and presentation form are each categorical variables. Each of these characteristics can only take a limited number of categorical values.

To test the correlation between the difficulty on the SCW training platform and the IRT difficulty, the coefficient of determination (R-squared) can be used, as both variables are numerical variables. The test (R $= 0.03$, $p = 1.85 \times 10^{-3}$) determines that no significant variation in the IRT difficulty can be explained by the SCW difficulty. This is no surprise, it is clear that the current difficulty assigned on the platform is not an accurate representation of the actual difficulty.

A one-way Analysis Of Variance (ANOVA) can be used for the categorical variables. This test determines if there is a statistically significant difference between one or more of the possible values that a categorical variable can take. The one-way ANOVA compares the mean difficulty for each of those values and determines whether any of those means are statistically significantly different from each other. If the one-way ANOVA returns a statistically significant result, that means there are at least two values that are statistically different from each other.

The variables were found to have unequal variances across the possible values, a property called heterogeneity of variances. A Welch ANOVA should hence be used, as this ANOVA can account for type I errors [77]. The results of the Welch ANOVAs are shown in Table 4.1 and show that a statistically significant difference exists between values for each of the categorical variables.

In this ANOVA, the eta-squared (η^2) metric was used to measure the effect size of each categorical variable on the estimated IRT difficulty. A medium effect was measured for the framework, category, and vulnerability of the exercise and a large effect was measured for the presentation.

Pairwise Games-Howell post-hoc tests were then used to determine which of the values in each categorical variable are statistically different from each other [78]. Results of these tests are shown in Table 4.2.

With the exception of the presentation, each of the categorical variables has a number of values that do not show a statistical significance with any other value. If there is no statistical significant difference between two values, this is usually due to insufficient data for one of the values or because the data of one (or both) of the values is too widely spread and there is significant overlap between the data for the

Table 4.1: The results of one-way ANOVA tests between variables of the exercises and the estimated difficulty. For each variable, the degrees of freedom (df) are shown, as well as the F-statistic, the p-value, and the eta-squared (η^2). All four categorical variables have statistically significant correlations with the estimated difficulty of the exercise. The presentation has a large effect on the estimated difficulty. The category, vulnerability, and framework each have a medium effect.

Variable	df	F	p	η^2
Category	36	14.154	2.97×10^{-53}	0.046
Vulnerability	142	5.393	1.19×10^{-45}	0.079
Framework	38	11.430	1.49×10^{-41}	0.060
Presentation	3	346.164	3.00×10^{-6}	0.155

Table 4.2: The second column displays the number of possible values. Every one of those values is paired with all other values for a pairwise Games-Howell test. The next two columns report the amount of pairs for which the means differ in a statistically significant way, and the amount of pairs for which this is not the case. In the final column the number of values is shown, for which at least one pair exists where the means are statistically significantly different.

Variable	Values	Significant pairs	Insignificant pairs	Significant values
Category	36	118	548	29
Vulnerability	142	384	9769	77
Framework	38	217	524	37
Presentation	3	3	0	3

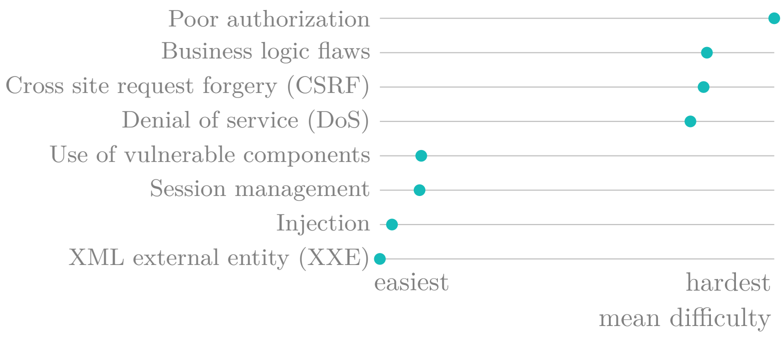


Figure 4.1: Vulnerabilities that often require only local changes to the code to fix have lower mean difficulties.

two values. The remaining values show a statistically significant difference in mean difficulty with at least one other value. When mean difficulties are compared in the remainder of this chapter, only values are compared whose means show statistically significant differences as determined by the pairwise Games-Howell tests. Each of the values that are compared explicitly, show statistically significant differences in mean difficulty with a p-value of at most 0.05. The exact p-values of these tests are reported in Appendix B.

4.1.3 Findings

In this section, I report some of the notable differences in difficulty between different values for each of the characteristics and offer possible explanations for these observations.

Vulnerability category

There is a statistically significant difference between the mean difficulty of the four hardest and the four easiest categories. When comparing these categories to each other, as shown in Figure 4.1, their difficulty seems mostly related to the locality of the vulnerability type.

The top four categories are (design) flaws, and the fix often involves larger pieces of code. This is especially the case for business logic flaws and denial of service, where the complexity of the code is the main cause of the problem. In these categories the developer is unable to foresee unintended results that are a direct result of the logic of their code.

The fixes of the four easiest categories usually only require local changes to the code. Injection and XML External Entity (XXE) are often fixed by properly sanitizing or whitelisting input parameters, or by configuring components properly. Security problems related to session management are things like incorrect session lengths, insufficient entropy, or insufficient length for the session ids. Use of vulnerable components is most frequently fixed by updating to a version of the component where the vulnerabilities are fixed. To fix these vulnerabilities, only local changes to the code are necessary, identifying the correct fix out of several options only involves building a mental model of a small piece of code, making it easier to reason about.

Still, it is noteworthy that the categories near the bottom of the scale are some of the most infamous vulnerabilities. Injection for example is the top category of the OWASP top 10 in 2017 and receives high scores for exploitability, detectability, and technicality. It has currently dropped to the third position in the iteration released in 2021[2]. Based on our training data, at least, it seems that these scores might be exaggerated. One exception to this is Cross-Site Scripting (XSS). Despite its infamy, it is closer to the middle of the scale, as can be seen in Figure 4.2 where XSS is shown together with all categories it shows statistically significant differences with. XSS being in the middle of the scale can be explained by the fact that there are two major types of XSS, stored XSS and reflected XSS. In the case of reflected XSS the vulnerability is rather local, and the fix is also applied locally, by using output encoding. For stored XSS there are usually multiple code fragments involved, one or more where the user input is stored as data and one or more where the stored data is used without output encoding.

In conclusion, the difficulty of the vulnerability correlates to the size of the related code fragments. Vulnerabilities that only require local changes in the code to fix are easier to understand and fix in training, despite their apparent prevalence in practice.

Framework

Of the 38 different frameworks, 37 show statistically significant differences with at least one other framework. Pseudocode shows a significant difference with 17 of these frameworks, all of which have a higher mean difficulty. This makes sense as pseudocode is an artificial language. It is designed to teach developers algorithms and other programming concepts, and should be easy to understand.

Memory management in memory-unsafe languages such as C and C++ can lead to a whole class of security problems that are avoided

[2] `https://owasp.org/Top10/`

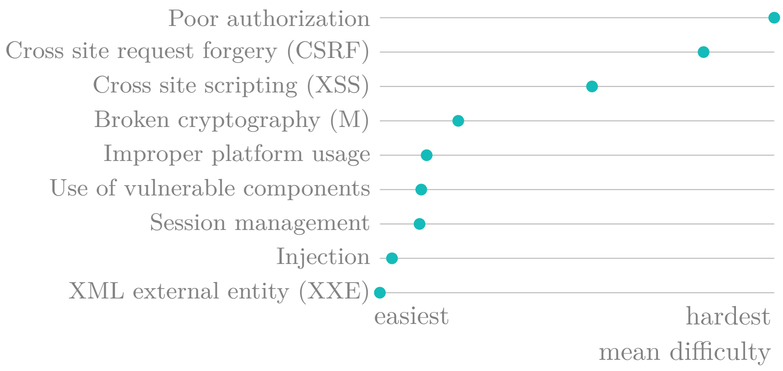


Figure 4.2: Challenges about XSS show a mean difficulty around the middle of the scale.

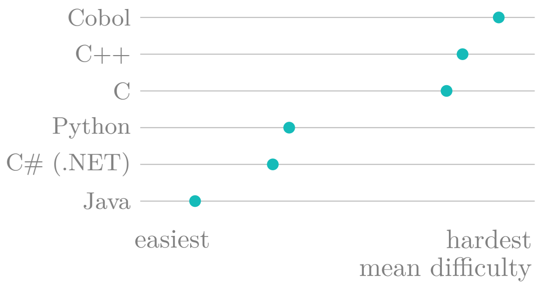


Figure 4.3: Older programming languages require memory management and use of pointers. They show harder mean difficulties than languages that have automated memory management.

in memory-safe languages. We can see this effect in the higher mean difficulty of C and C++ compared to those of the modern, memory-safe programming languages Java, C# (.NET), and Python, as shown in Figure 4.3. However, the memory-safe language Cobol also shows a statistically significantly higher mean difficulty compared to each of these three languages. While Cobol is memory-safe, it does still require memory management and use of pointers, which might explain the higher mean difficulty. Cobol is also known for its lack of clear documentation regarding security concepts.

Several frameworks show statistically significant differences between the framework and its standard programming language. This is the case for Java Spring, Java EE, Java JavaServer Faces (JSF), C# (.NET) Web Forms, and Python Django. All of these frameworks are more difficult than their standard language counterparts, as shown in Figure 4.4. This

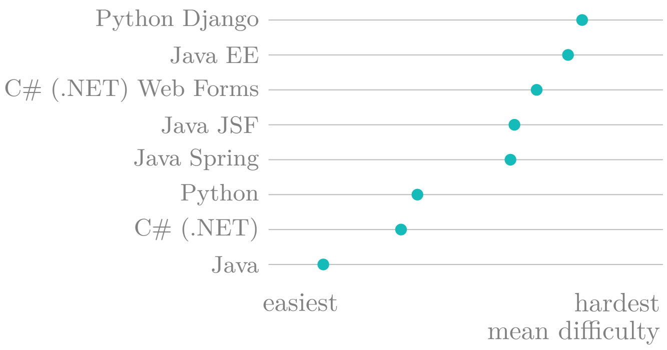


Figure 4.4: All of the frameworks that show statistically significant differences with their standard programming language, have a higher mean difficulty.

is surprising, as frameworks are designed to implement commonly used functions so that the developer does not have to. For example, to securely hash a password in standard Java, the developer has to research which algorithm is the most secure, they have to generate a salt using a secure random number generator and correctly combine each of these techniques. In Java Spring a `PasswordEncoder` interface is provided, the documentation of this interface is brief and informs developers that `BCryptPasswordEncoder` is the preferred implementation[3].

Because frameworks often automate these commonly used features, some details of the implementations might be lost to developers. This is also confirmed by previous research where 44% of analyzed applications were shown to contain vulnerabilities caused by misunderstandings of the framework implementation details [79]. When the implementation in such a framework is insufficient, or when the framework is used incorrectly, it is possible that even a security conscious developer can remain unaware of the consequences. On top of the standard language skills, and the security knowledge, a developer using a framework hence also needs an intimate knowledge of the framework itself to deliver secure code.

The mobile framework Java Android is more difficult than web frameworks for the same language (Java JSF, Java Spring), as shown in Figure 4.5. This can be explained through the increased attack vectors of mobile applications that are installed on the device of the user. Because the device and operating system cannot be trusted, the developer has to

[3]`https://docs.spring.io/`

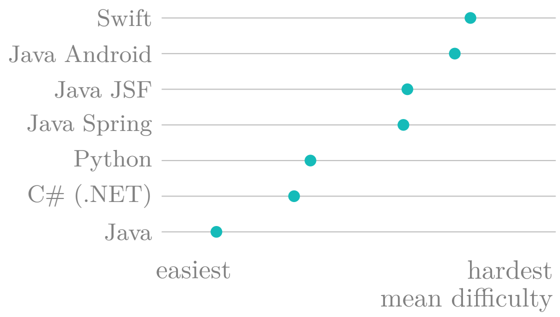


Figure 4.5: Mobile framework and languages that show statistically significant differences with their web counterparts have higher mean difficulties.

be aware of other threats, such as restoring backups that are tampered with, detecting root access, and tapjacking, a vulnerability where the attacker can trick the user to perform unintended actions by drawing overlays. Unfortunately, Objective C, an extension of C that can be used for mobile development, does not show a statistically significant difference with C itself to confirm this correlation. The more modern replacement for Objective C, Swift, that is inspired by both C# (.NET) and Python does show a statistically significant higher mean difficulty than both these languages [80].

Languages and frameworks have evolved to automate some of the tasks of the developer, such as memory management, and password encryption. The abstractions provided by these languages and frameworks do not always have a positive impact on security, as is evident from these results. It is likely that the developer has to sufficiently grasp the implementation details to fully understand the security impact of their code.

Presentation

A challenge can be presented to the user as an identify, locate, or fix exercise. It is unsurprising that identifying a vulnerability that is already marked in the code, is the easiest type of exercise, as shown in Figure 4.6. On the platform this type of exercise is usually the first stage of a two-stage challenge, with the second stage a fix exercise.

According to the 2PL model, locating vulnerabilities in the code is the most difficult task. This is in line with observations made in practice, where many vulnerabilities go unnoticed by developers and are detected

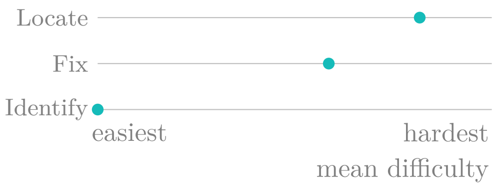


Figure 4.6: The mean difficulty of each presentation form is statistically different of the other two. In line with observations in practice, locating a vulnerability in code is the most difficult of the three tasks, followed by fixing the vulnerability.

by automated tools at later stages in the Software Development Life Cycle (SDLC).

4.2 Step size adjustment ability estimation

From the previous experiment in Section 4.1, it is evident that calibrating the entire item bank takes a long time. Similarly, computing the ability estimate based on the entire response pattern of a user takes too long. Currently, about 15,000 challenges are completed every day, that is about one challenge every six seconds and this number is only expected to go up. The computations to estimate the ability of a single user frequently exceed that. For a procedure that has to be executed this often, even a minute is too long.

4.2.1 Goals and research questions

The main *goal* of this experiment is to evaluate different approximation procedures for the ability estimates. The *purpose* is to determine if they can be used to improve the efficiency of the ability computations without a significant loss in accuracy. The *quality focus* is the accuracy of the procedures with increasing numbers of answered items. The above goal can be achieved by means of an experiment aimed at answering the following question for each procedure:

- **Q1** how big is the mean error of the approximation after every 5 challenges?

4.2.2 Approximation procedures

In research literature, I have found two so-called step size adjustment procedures that are used in CATs. These procedures update the latest estimate based on either a fixed or variable step size [70].

Fixed step size With a fixed step size, the ability estimate is increased (or decreased) by a specific amount, often between 0.4 and 0.7, when the user answers an item correctly (or incorrectly). In this experiment the smallest step size of 0.4 is used for the evaluation.

Variable step size With a variable step size the new ability estimate is placed at the halfway point between the current estimate and the difficulty of one of the two most extreme items in the item bank. This is possible because the calibration techniques of IRT place users and items on the same scale. If the user answers an item correctly, then the highest item parameter is used, if not, the lowest is used. This procedure makes sense when one considers that the item selection algorithm in CATs continuously selects items that are significantly above the current estimated ability level of the user, as explained in Section 3.2.3. This is not the case for the historical data, and will also not be the case for the item selection of the ITS in the future. With more forgiving item selection this procedure will likely become inaccurate over time.

Adaptive step size As an improvement, I have developed a variation of this procedure that uses the difficulty of the selected item instead of the difficulty of the extreme items in the item bank. This adaptive step size procedure is shown in Algorithm 3. When the user answers an item correctly that was expected to be hard, the ability estimate is increased to the halfway point between this item and the current estimate. Similarly, when the user answers an item incorrectly that was expected to be easy, the ability estimate is decreased to the halfway point.

In the other cases, the outcome of the answer confirms that the current ability estimate is accurate. A question that is more difficult than the user's current ability level is answered incorrectly, or a question below the ability level is answer correctly. The player can then optionally be rewarded or punished with a fixed value, similar to the fixed step size adjustment procedure. For this experiment, two variations of the adaptive step size procedure are tested, one with a fixed reward of 0.2 and one without a fixed reward.

Algorithm 3: Adaptive step size adjustment procedure

```
input  : user ability $\boldsymbol{\theta}_i$
         item difficulty $\beta_j$
         answer $X_{ij}$
         an optional punishment/reward value $r$
output: updated user ability $\boldsymbol{\theta}_i$
1  if $X_{ij}$ is correct then
2      if $\boldsymbol{\theta}_i \leq \beta_j$ then
3          return $(\boldsymbol{\theta}_i + \beta_j)/2$
4      else
5          return $\boldsymbol{\theta}_i + r$
6  else
7      if $\boldsymbol{\theta}_i \geq \beta_j$ then
8          return $(\boldsymbol{\theta}_i + \beta_j)/2$
9      else
10         return $\boldsymbol{\theta}_i$ - $r$
```

4.2.3 Experimental set-up

The different procedures are evaluated by comparing their approximations with the (accurate) estimated IRT ability. To do this, an IRT ability estimate is needed for each user at each point in time, which requires many long-running IRT calibration procedures. The five frameworks with the most data were chosen to use in the evaluation. These are Java Spring, Java EE, NodeJS Express, Pseudocode and Python Django. The IRT ability was estimated for every user after every 5 completed challenges.

Each of the four procedures starts from the IRT ability estimate after 20 completed challenges. From that point forward, the approximation methods are applied to the outcome of each challenge attempt. The resulting approximations are compared to the IRT ability after every 5 challenges.

4.2.4 Findings

For each approximation method, the evolution of the mean error as a percent of the full ability scale, is shown in Figure 4.7. The fixed step size adjustment becomes excessively inaccurate after only 15 to 20 challenges following the initial calibration. After 200 challenges, the mean error of this approximation is up to 300%. While the adaptive step size procedure with a fixed reward is significantly better, its error still becomes excessively large and rises indefinitely. The seemingly ever increasing error for these two procedures is explained by the large amount of users who play challenges that are below their skill level, and hence often answer them correctly. With these two procedures, the ability level of users like this is increased each time, while these answers have no significant impact on the accurate estimates.

This effect is still visible for the variable step size adjustment procedure. However, with this procedure the estimate does not exceed the difficulty of the most difficult item in the item bank, which limits the error to about 30%.

The adaptive step size procedure without reward or punishment is the most accurate. With this procedure, the ability estimate is not adjusted if the outcome of an item is as expected according to the current estimate. The ability estimates of users who are answering easy items correctly is not updated. For correct answers to difficult challenges or for incorrect answers to easy challenges, the estimate is moved in the right direction, but using smaller steps than the variable step size procedure. The mean error of 10% is acceptable for it to be used in the ITS, as will become evident in a following experiment in Section 4.4. This implies that the full calibration procedure is only necessary once, for the initial calibration. In practice, the abilities for existing users will still periodically be re-calibrated along with the initial calibration of new users. The adaptive step size procedure itself takes roughly 500 ns to estimate the new ability level.

4.3 Collaborative filtering algorithms

4.3.1 Goal and research questions

The main *goal* of this experiment is to test the prediction accuracy of different (variations of) CF algorithms. The *purpose* is to determine which algorithms are most effective at making recommendations on the SCW platform. The *quality focus* is the error of the predicted ratings compared to the observed ratings. The above goal can be achieved by means of an experiment aimed at answering the following questions:

- **Q1** What is a good error rate for this data set?
- **Q2** Which algorithm achieves the lowest error rate for predicting the utility of an item?

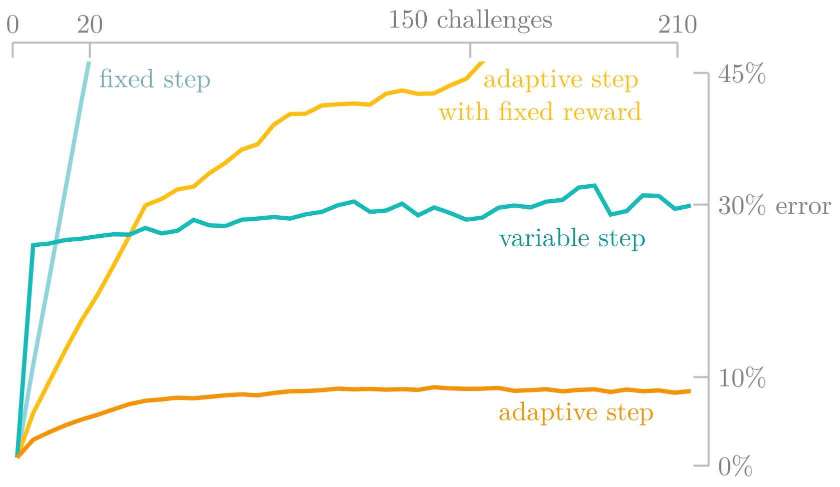


Figure 4.7: The error rates of the two procedures with a fixed step or a fixed reward become excessively large. For the variable step size procedure, the error rate is capped at around 30%. The adaptive step size procedure without fixed reward is the most accurate and its error rate does not exceed 10%.

4.3.2 Benchmark algorithms

To train and evaluate different (variations of) algorithms, I have used the open-source Python scikit Surprise [36]. Since it is open source, the code is available to download from GitHub and the implemented algorithms can later be adapted to learning systems.

Surprise provides two basic CF algorithms that can be used as benchmarks to answer the first research question.

The first benchmark algorithm is called Normal Predictor, it estimates a normal distribution based on the training data and makes new predictions by randomly sampling from this distribution.

The second algorithm is the baseline algorithm. It uses the baseline estimate as defined in Equation 4.2 [37].

$$\hat{r}_{ui} = b_{ui} = \mu + b_u + b_i \tag{4.2}$$

In this equation, μ is the observed overall mean of the ratings, and b_u and b_i are the biases, the average observed deviations of this mean by user u and item i.

When computing the utility of an item in Section 3.4.3 it was observed that some users show a consistent bias. This is because many users on the SCW platform stick to the predetermined courses or tournaments. For users who have a higher ability level, this current selection of challenges is consistently too easy, leading to boredom and hence consistently lower ratings by these users. The utility experienced by these users can likely be predicted relatively accurately using the baseline algorithm. Hence it is expected that this benchmark algorithm will already perform well.

4.3.3 Memory-based algorithms

Five memory-based CF algorithms are evaluated in this experiment, four k-NN algorithms and an algorithm called the Slope One algorithm.

The k-NN algorithms predict the rating $\hat{r}_{ui}$ of a user u for an item i based on the observed ratings r_{vi} of similar users v for that item. These algorithms explicitly combine the ratings of the k nearest neighbours to compute a prediction, hence their name as k-NN algorithms. The difference between these algorithms lies in the exact formulas used to combine the existing ratings of the neighbours to make a prediction.

k-NN basic To predict a rating, the first k-NN algorithm, k-NN basic, takes the weighted average of the observed ratings for that item by the k nearest neighbours of the user. As shown in Equation 4.3, the similarity between the users is used as the weight in the weighted average.

$$\hat{r}_{ui} = \frac{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v) \cdot r_{vi}}{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v)} \tag{4.3}$$

In this equation, $\text{sim}(u,v)$ is the similarity between users u and v, and $N_i^k(u)$ is the set of k nearest neighbours of user u that have rated item i. Different similarity metrics can be used, as will be explained later in this section.

The algorithm can also be used to do item-based CF, by summing over $j \in N_u^k(i)$, the k nearest neighbours of item i that are rated by user u. In that case, the similarity between items $\text{sim}(i,j)$ needs to be used.

k-NN with means The second k-NN algorithm is called k-NN with means. It does not take the weighted average of the ratings of the nearest neighbours, but instead uses the deviation of the mean, as shown in Equation 4.4.

$$\hat{r}_{ui} = \mu_u + \frac{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v) \cdot (r_{vi} - \mu_v)}{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v)} \tag{4.4}$$

In this equation, μ_u is the mean rating given by user u. The algorithm can similarly be used in an item-based fashion.

k-NN with z-score The k-NN with z-score algorithm uses the standard score, or z-score of the ratings of each user. The standard score is the number of standard deviations the rating deviates from the mean. It can be computed by taking the deviation from the mean, like in the previous equation, and divide the result by the standard deviation. The resulting formula is shown in Equation 4.5.

$$\hat{r}_{ui} = \mu_u + \sigma_u \frac{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v) \cdot (r_{vi} - \mu_v)/\sigma_v}{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v)} \tag{4.5}$$

In this equation, σ_u is the standard deviation of the ratings of user u. This algorithm can also be trivially changed to make item-based predictions.

k-NN baseline The final k-NN algorithm, k-NN baseline, uses the baselines of the k nearest neighbours to make a prediction, as shown in Equation 4.6

$$\hat{r}_{ui} = b_{ui} + \frac{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v) \cdot (r_{vi} - b_{vi})}{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v)} \tag{4.6}$$

In this equation, b_{ui} is the baseline rating of user u for item i, as computed in Equation 4.2.

Similarity metrics The k-NN algorithms described above all make use of the similarity between users (or items) to make predictions, in the formulas this similarity between two users u and v is denoted as $\text{sim}(u, v)$. Four similarity metrics are considered in this experiment.

The default configurations for the k-NN algorithms use the *Mean Squared Difference (MSD) similarity.* The MSD is a metric for distance between two users, it sums over all items that have been rated by both users and takes the square of the difference of the given ratings, as shown in Equation 4.7.

$$\text{msd}(u, v) = \frac{1}{|I_{uv}|} \cdot \sum_{i \in I_{uv}} (r_{ui} - r_{vi})^2 \tag{4.7}$$

In this equation, I_{uv} is the set of items that have been rated by both users u and v.

To use this distance as a similarity measure, the inverse has to be taken. To avoid dividing by zero, a one is added to the denominator, as shown in Equation 4.8.

$$\text{msd\_sim}(u, v) = \frac{1}{\text{msd}(u, v) + 1} \tag{4.8}$$

The second metric, *the cosine similarity*, requires the ratings of the common items between two users to be represented as vectors. The cosine similarity is then defined as the cosine of the angle between these two vectors, which is the same as the inner product of the two vectors normalized to both have length one. The resulting formula is shown in Equation 4.9.

$$\text{cosine\_sim}(u, v) = \frac{\sum\limits_{i \in I_{uv}} r_{ui} \cdot r_{vi}}{\sqrt{\sum\limits_{i \in I_{uv}} r_{ui}^2} \cdot \sqrt{\sum\limits_{i \in I_{uv}} r_{vi}^2}} \tag{4.9}$$

The third metric is *the Pearson correlation coefficient*. It is commonly used in statistics as a measure of correlation between two data sets. It can be seen as a mean-centered cosine similarity which is evident from similarity of the formula with that of the cosine similarity, as shown in Equation 4.10.

$$\text{pearson\_sim}(u, v) = \frac{\sum\limits_{i \in I_{uv}} (r_{ui} - \mu_u) \cdot (r_{vi} - \mu_v)}{\sqrt{\sum\limits_{i \in I_{uv}} (r_{ui} - \mu_u)^2} \cdot \sqrt{\sum\limits_{i \in I_{uv}} (r_{vi} - \mu_v)^2}} \tag{4.10}$$

Baseline, the final similarity metric considered in this experiment, is similar to the Pearson correlation coefficient, but uses baselines for centering instead of means. This similarity metric is shown in Equation 4.11.

$$\text{pearson\_baseline\_sim}(u,v) = \frac{\sum\limits_{i\in I_{uv}} (r_{ui} - b_{ui}) \cdot (r_{vi} - b_{vi})}{\sqrt{\sum\limits_{i\in I_{uv}} (r_{ui} - b_{ui})^2} \cdot \sqrt{\sum\limits_{i\in I_{uv}} (r_{vi} - b_{vi})^2}} \tag{4.11}$$

Slope One To predict a rating $\hat{r}_{ui}$ of a user u for an item i, k-NN algorithms only look at the ratings by similar users for that same item r_{vi}. The ratings of other common items are only used to determine the most like-minded users, not to compute the predicted rating.

In the Slope One algorithm, all ratings of common items are used to make a prediction by computing a popularity differential between each pair of items [81]. To compute this popularity differential between two items for two users, the rating of one user for this item is subtracted from the rating of the other [36]. This can be done for all users to receive an average popularity differential between these two items, as shown in Equation 4.12.

$$\text{popularity\_differential}(i,j) = \frac{1}{|U_{ij}|} \sum_{u\in U_{ij}} r_{ui} - r_{uj} \tag{4.12}$$

In this equation, U_{ij} is the set of all users who rated both items i and j.

This popularity differential can then be used to make predictions for the ratings, by taking the mean rating of a user and adding the average popularity differential between two items to this mean, as shown in Equation 4.13.

$$\hat{r}_{ui} = \mu_u + \frac{1}{|R_i(u)|} \sum_{j\in R_i(u)} \text{popularity\_differential}(i,j) \tag{4.13}$$

With $R_i(u)$ the set of items j that have been rated by u and also have been rated by at least one other user that has rated i.

4.3.4 Model-based algorithms

Besides these memory-based algorithms, Surprise also offers implementations for five popular model-based algorithms [36], one based on clustering and four based on matrix factorization.

Co-Clustering The discussed memory-based algorithms make predictions based on a neighbourhood of either like-minded users or similarly-rated items. In the Co-Clustering CF algorithm, the idea is to simultaneously obtain user and item neighbourhoods via co-clustering [43]. Many algorithms exist to assign these co-clusters, sometimes called bi-clusters [82]. The implementation in Surprise is a more straightforward optimization method, similar to k-means [36].

The average ratings from these clusters can then be used to make predictions about the ratings, as shown in Equation 4.14.

$$\hat{r}_{ui} = \overline{C_{ui}} + (\mu_u - \overline{C_u}) + (\mu_i - \overline{C_i}) \tag{4.14}$$

In this equation, $\overline{C_{ui}}$ is the average rating of co-cluster C_{ui}, $\overline{C_u}$ is the average rating of the cluster of user u, and $\overline{C_i}$ is the average rating of the cluster of item i.

Probabilistic Matrix Factorization (PMF) Matrix factorization models map both the users and the items to a space of latent factors of dimensionality f. This latent space tries to explain ratings by characterizing both users and items. For example, on the SCW training platform, factors might be obvious characteristics of items such as the vulnerability category, or the presentation of the item. It is also possible that they represent less defined dimensions such as readability of the code, or the structure of the files, or even completely uninterpretable dimensions [38].

Each item i is then depicted as a vector $q_i \in \mathbb{R}^f$, that measures the extent to which this item possesses the characteristics represented in the latent space. Each user u is represented by a vector $p_u \in \mathbb{R}^f$ that measures the extent of the interest (or utility) this user has for the corresponding factors.

The dot product of these two vectors captures the interest (or utility) of the user for this item and can be used to make predictions, resulting in the PMF algorithm [36, 44]. The formula is shown in Equation 4.15.

$$\hat{r}_{ui} = q_i^T p_u \tag{4.15}$$

Singular Value Decomposition (SVD) The SVD algorithm is similar to PMF, but creates a final rating by also adding the baseline predictors [36, 38, 83]. The resulting formula is shown in Equation 4.16.

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u \tag{4.16}$$

To estimate each of these unknown variables, the following regularized squared error needs to be minimized:

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 \right) \tag{4.17}$$

The constant λ controls the extent of regularization, in this experiment it is set to 0.02. Minimization of the error is typically done using stochastic gradient descent [36, 38]. In this process, the algorithm makes predictions for all existing ratings and computes the prediction error. All parameters are then moved slightly in the opposite direction to improve the predictions in the next pass. For the algorithm in this experiment, this process was repeated 20 times to reach the final estimates.

Non-negative Matrix Factorization (NMF) The NMF CF algorithm is also similar to PMF, but the user and item factors are kept positive [36, 84–86]. That means each item i is depicted as a vector $q_i \in \mathbb{R}^f_{\geq 0}$, and each user u as a vector $p_u \in \mathbb{R}^f_{\geq 0}$. Predictions are then computed identically to PMF, as shown in Equation 4.15.

To ensure that the parameters remain positive, a stochastic gradient descent is used that ensures non-negativity of factors based on the step size and the initial values [36].

SVD++ The SVD++ algorithm was developed to make use of implicit feedback. It builds on the assumption that the users can also be characterized by accounting for which items they have or have not provided a rating for. The SVD++ algorithm often results in superior accuracy compared to SVD.

A second set of item factors $y_i \in \mathbb{R}^f$ is added that is used to characterize the users based on the set of items they have rated. The user vector p_u is then extended based on these characteristics before using it to compute a prediction, as shown in Equation 4.18.

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left(p_u + |I_u|^{-\frac{1}{2}} \sum_{j \in I_u} y_j \right) \tag{4.18}$$

4.3.5 Experimental set-up

To evaluate the various CF algorithms in this section, I have selected the five frameworks with the most data available on the platform. These are Java Spring, Java EE, NodeJS Express, Pseudocode and Python Django. For each of those frameworks, I have iterated over the challenge attempts in the data set and continuously estimated the ability of each user. This estimate is computed with the adaptive step size adjustment procedure, evaluated in the previous experiment of this chapter. This ability estimate was then used to rate each challenge attempt according to its utility for the user, as described in Section 3.4.3. The resulting ratings, on a scale of 1 to 5, are used to evaluate the (variations of) CF algorithms.

- The two benchmark algorithms are tested.
- All four k-NN algorithms are tested with default configurations.
- Each k-NN algorithm is tested with item-based similarities.
- Each of the four k-NN algorithms is tested with all four similarity metrics.
- Slope One is tested with default configurations.
- All five model-based algorithms are tested with default configurations.

Each (variation of an) algorithm is evaluated on each of the five selected frameworks using a process of 10-fold cross-validation. In this process, the data set of the framework is split into 10 equal parts, called folds. Next, 9 of the 10 folds are used as training data, and the final fold is used as test data. Every time an algorithm is evaluated on a fold of test data, this is done with three different evaluation metrics that will be explained further in this section. This process is repeated 10 times, each time alternating the fold to be used as test data. So for each algorithm and framework combination, 10 measurements are obtained, one for each fold. The final step of the cross-validation process is to take the mean and standard deviation of these 10 measurements.

The mean of the 10 measurements is used to evaluate the algorithm on this particular data set, as it represents the mean performance of the algorithm. I will call this the framework-mean μ_f for an algorithm. The set of five framework-means for an algorithm is then denoted as M_f. The standard deviation can be used to assess the algorithm for overfitting. Overfitting is a prediction error caused when the algorithm is too closely aligned to the training sets, causing decreased performance on (some of) the test sets. Analogous with the framework-mean, this will be denoted as $\sigma_f \in S_f$.

If all five standard deviations $\sigma_f \in S_f$ are sufficiently small, we can take the mean of the five framework-means $\mu_f \in M_f$ to obtain a final result that enables us to evaluate the algorithm's performance on all five frameworks. This mean will be called the algorithm-mean μ_a. In Appendix C, the algorithm-mean is reported for every (variation of) algorithm evaluated, as well as the largest of the framework standard deviations $\sigma_{max} = \max(\sigma_f \in S_f)$ to show that no overfitting is taking place.

Metrics

To evaluate the CF algorithms, I have used three metrics that compare the set of predicted ratings $\hat{R}$ to the set of observed ratings R.

Mean Absolute Error (MAE) One of the most popular metrics in research literature is the MAE [35, 87]. It computes the average prediction error over the entire set of predictions, as shown in Equation 4.19.

$$\text{MAE} = \frac{1}{|\hat{R}|} \sum_{\hat{r}_{ui} \in \hat{R}} |r_{ui} - \hat{r}_{ui}| \tag{4.19}$$

The MAE uses the same scale as the ratings itself, and hence cannot be used to make comparisons between measurements using different scales. For this the Normalized Mean Absolute Error (NMAE) can be used that expresses the error as a percentage of the full scale [35]. In our rating scale, the MAE takes values between 0 for a perfect prediction and 5 as the maximum error.

Root Mean Squared Error (RMSE) The RMSE amplifies the error between the actual rating and the predictions by taking the square of the prediction error, as shown in Equation 4.20.

$$\text{RMSE} = \sqrt{\frac{1}{|\hat{R}|} \sum_{\hat{r}_{ui} \in \hat{R}} (r_{ui} - \hat{r}_{ui})^2} \tag{4.20}$$

Like the MAE, the RMSE is scale-dependent. It takes non-negative values and a lower value means better prediction performance.

The RMSE has increased in popularity, partly because of its use in the Netflix competition for movie recommendations [35, 88]. In 2006, Netflix launched a competition to beat the current score of their recommendation algorithm, *Cinematch.* At the time the algorithm achieved a

RMSE of 0.9514 [88, 89]. With the competition, Netflix offered a $1 million dollar prize to the team that could improve this benchmark by 10%. The algorithm of the winners, one year later, reached a RMSE of 0.8567 and was put into production [88, 90, 91]. This also shows that in CF algorithms, improvements of several percents can already be valuable.

Fraction of Concordant Pairs (FCP) Finally, I have also included a measure that puts more focus on the order the items are ranked rather than the exact rating that is predicted. In the FCP, the number of concordant pairs n_c^u for a user u is determined by counting the pairs of ratings that are ranked correctly [92]. This is shown in Equation 4.21.

$$n_c^u = |\{(i,j) \mid \hat{r}_{ui} > \hat{r}_{uj} \text{ and } r_{ui} > r_{uj}\}| \tag{4.21}$$

The number of pairs that is ranked incorrectly is the number of discordant pairs n_d^u.

The total number of concordant pairs n_c and discordant pairs n_d is then obtained by summing over all users, as shown in Equation 4.22.

$$n_c = \sum_u n_c^u, \qquad n_d = \sum_u n_d^u \tag{4.22}$$

Using this total number of concordant and discordant pairs, the FCP can be computed as shown in Equation 4.23.

$$\text{FCP} = \frac{n_c}{n_c + n_d} \tag{4.23}$$

The FCP takes values between 1 and 0, with higher meaning that a larger portion of pairs are ranked correctly.

4.3.6 Findings

All measurements during this experiment are included in the tables of Appendix C. In this section, I will discuss some interesting results, and mainly use the MAE to compare the performance of algorithms.

Benchmark

The benchmark algorithms perform reasonably well. The Baseline algorithm in particular, as it reaches a μ_a MAE of 0.5004 and μ_a RMSE of 0.6169. This is significantly lower than the prediction error for the Netflix prize for example, where the winner reached a RMSE of 0.8567 [88, 90, 91]. The Netflix recommendations happen to use the same scale as the utility rating in this experiment, so this comparison can be made. These results indicate that the utility of an item on the SCW platform is in comparison easier to predict.

The good performance of the Baseline algorithm was expected based on the observations of the data. In the current item selection, most users are given the same exercises. These items are of course not randomly selected, but carefully tailored by the SCW employees who designed these OWASP Top 10 courses on the platform. As a result, for a number of users this item selection is rather good, and the ratings of these users are consistently high. For more experienced users, the content of these courses is too easy, and as a result their ratings are consistently lower. Using baselines, these consistent bias result in reasonably accurate predictions.

Memory-based algorithms

The default configurations of all memory-based algorithms result in improved prediction performance compared to the baseline benchmark. The Slope One algorithm performs worse than the k-NN algorithms. It reaches a MAE of 0.4928 which is an improvement of about 1.5% compared to the baseline benchmark. The best performing k-NN algorithm is the k-NN baseline. As explained before, it is not surprising that baseline algorithms are performing well on this data set. It reaches a MAE of 0.4680, improving the baseline benchmark by 6.5%.

In the next test, the k-NN algorithms were configured to use item-based similarities to find the k nearest neighbouring items, and make predictions based on the ratings of those items. This item-based configuration performed worse for all four algorithms, with an increase in prediction error between 1.0% and 6.7% depending on the algorithm.

In the final test with the k-NN algorithms, each of the algorithms is evaluated in combination with each of the four similarity metrics. The results indicate that the similarity metric can have a big impact on the prediction performance, with as much as a 7% increase in performance for the k-NN with z-score algorithm between the cosine similarity metric and the baseline similarity metric.

When the similarity metrics are ranked from best to worst performance, this ranking is the same for each algorithm, and is as follows:

1. Baseline similarity,
2. Cosine similarity,
3. MSD similarity,
4. Pearson similarity.

Table 4.3: All memory-based algorithms perform better than the benchmarks. k-NN basic and Slope One perform significantly worse than the other algorithms.

	MAE	
	μ_a	σ_{max}
k-NN basic	0.4902	0.003
k-NN with means	0.4521	0.002
k-NN with z-score	0.4508	0.002
k-NN baseline	0.4514	0.002
Slope One	0.4928	0.002

Once again, the choice based on baselines is the best performing.

For each of the k-NN algorithms, the item-based similarity, computed with the baseline similarity metric, results in the best performance. The MAE of the best performing configuration for each memory-based algorithm is shown in Table 4.3, all other measurements are available in Appendix C.

Model-based algorithms

In the next test, these results are compared to those of the model-based algorithms, which tend to be more accurate, especially for sparse data.

Surprisingly, none of the model-based algorithms was able to improve the prediction rate of the best performing memory-based algorithm. Changing parameters in the model-based algorithms had influence on the speed of convergence, but no real impact on the prediction accuracy. The MAE for each algorithm is shown in Table 4.4, for other measurements consult Appendix C.

SVD++, the best performing of the model-based algorithms, reaches a MAE of 0.4591, beating the default configurations of all memory-based algorithms. This is not surprising, as SVD++ also takes into account which items users have and have not rated. This algorithm can hence make a better distinction between users who followed the standard courses, and users who did not.

Co-Clustering performs the worst of the model-based algorithms and its MAE is not significantly better than that of the baseline benchmark. One of the strengths of the Co-Clustering algorithm is that it can handle synonyms rather well. In recommendation systems, synonyms are items that are (nearly) identical but are still labeled as different items. Movie databases often have genres like “Children’s movie” and “Children’s film”, which are then clustered together by the algorithm. More

Table 4.4: Most model-based algorithms perform worse than the fine-tuned memory-based algorithms. SVD++ is the exception, it reaches prediction accuracy close to that of the best performing memory-based algorithms.

	MAE	
	μ_a	σ_{max}
Co-Clustering	0.4999	0.007
PMF	0.4783	0.003
NMF	0.4835	0.002
SVD	0.4750	0.003
SVD++	0.4591	0.003

data can then be used to make recommendations about either of those genres. Evidently, on our platform few such synonyms are present, even though for each framework many exercises exist about the same vulnerability types. The ratings for these exercises are sufficiently inconsistent so that the Co-Clustering algorithm does not result in significant improvements in prediction accuracy.

While model-based algorithms are usually more accurate, this is not the case for the SCW training data. This is likely because the advantages of model-based algorithms are not very applicable to this data set, as will be explained in the discussions in Chapter 5.

4.4 Adaptation to learning systems

Through the above experiments, I am able to obtain an accurate difficulty measure for each challenge, and a good approximation of the ability level of each user at every point in time. I have also rigorously tested the performance of several CF algorithms on the data set. In the experiment of this section, I will use the ability level of the users to improve the performance of these algorithms.

4.4.1 Goals and research questions

The main *goal* of this experiment is to test the prediction accuracy of different CF algorithms after they have been adapted to learning systems. The *purpose* is to determine if the proposed adaptations in this work lead to improved performance in some or all algorithms. The *quality focus* is the prediction error of the adapted algorithms in comparison to those of the unaltered algorithms. The above goal can be achieved by means of an experiment aimed at answering the following question for each algorithm:

- **Q1** Does the error rate of the prediction decrease when the proposed adaptation to learning systems is applied?

4.4.2 Experimental set-up

The best performing configuration of each algorithm tested in the previous experiment of Section 4.3 has been adapted to learning systems and measured in this experiment.

The same process of evaluation is used as in the previous experiment, where each algorithm is tested using 10 fold cross-validation on each of the top five frameworks. For each algorithm, the algorithm mean μ_a and the largest standard deviation σ_{max} of each of the three metrics are reported in Appendix C.

4.4.3 Adaptation to learning systems

The goal of the adaptation is to include the ability level θ_{ui} of user u at the time of rating item i in the algorithm. In particular, this ability level should be used to update the similarity between users. Currently, two users are considered like-minded if they rate common items similarly. With this adaptation, the goal is to consider two users like-minded if the items they rated around the same ability level are rated similarly.

Similarity metrics

The most obvious way to include this ability level is to update the similarity metrics used by the k-NN algorithms.

In each of the metrics, to determine the similarity between users u and v, summations are made over I_{uv}, the items that have been rated by both users u and v. In the adaptation, this set is replaced by I^t_{uv}, the items that have been rated by both users u and v and for which the difference in ability level $\delta\theta = |\theta_{ui} - \theta_{vi}|$ at the moment of rating is smaller than the threshold t.

In Equation 4.24, this adaptation is demonstrated for the Baseline similarity, the similarity metric that resulted in the best performance in the previous experiment.

$$\text{baseline\_adapted\_sim}(u,v) = \frac{\sum\limits_{i \in I_{uv}^t} (r_{ui} - b_{ui}) \cdot (r_{vi} - b_{vi})}{\sqrt{\sum\limits_{i \in I_{uv}^t} (r_{ui} - b_{ui})^2} \cdot \sqrt{\sum\limits_{i \in I_{uv}^t} (r_{vi} - b_{vi})^2}} \tag{4.24}$$

The value of the threshold parameter t needs to be large enough so that there is still enough data within this range to compare users with. The best performance was measured when t was around a third of the entire ability scale.

Data processing

The other algorithms do not explicitly use similarity between the users. They can still be influenced to take the ability level at the time of rating into account by processing the data first.

Instead of using a filter at the time of computing the similarity, it is possible to filter at the time the rating is given. To do this, the ability scale is split into three intervals, called expert, intermediate, and novice ability level.

The ability level of the user at the time of rating an item i is taken into account, by splitting the item into three distinct items, i_{expert}, $i_{intermediate}$, and i_{novice}. If the user rates the item i, the data is then processed as rating one of these three items, based on the ability level of the user at the time of rating. This processed data can then be used to train the unaltered algorithms.

This is less accurate than adapting the similarity metrics. It is possible that users at the bottom of the expert ability level rate items similarly to the users at the top of the intermediate ability level. By using a threshold at the time of computing the similarity, these users are still considered similar. By splitting the ability scale into ranks, they are not.

4.4.4 Findings

The proposed adaptation resulted in improved performance for all of the algorithms, as shown in Table 4.5. The improvement is larger for the k-NN algorithms, because these algorithms make more explicit use of the ratings of neighbours to make a prediction. The improvement is particularly large for the k-NN basic algorithm, with a 13.7% decrease

Table 4.5: All algorithms have improved performance because of the adaptation to learning systems.

	MAE		
	μ_a	σ_{max}	
	Similarity metric		
k-NN basic	0.4232	0.002	-13.7%
k-NN with means	0.4261	0.002	-5.7%
k-NN with z-score	0.4276	0.002	-5.1%
k-NN baseline	0.4206	0.003	-6.8%
	Data processing		
Slope One	0.4719	0.002	-4.2%
Co-clustering	0.4940	0.007	-1.2%
PMF	0.4598	0.003	-3.9%
NMF	0.4614	0.003	-4.6%
SVD	0.4555	0.002	-4.1%
SVD++	0.4409	0.003	-4.0%

in MAE. This algorithm is straightforward as it takes the mean of the ratings of the most similar users to make a prediction. The improvement of this algorithm is a clear indication that the ability level of a user at the time of rating an item has a significant impact on the rating.

The smallest improvement is made by the Co-Clustering algorithm. This algorithm was already the worst-performing, and it is likely that the effect of splitting the items is somewhat negated by the clustering.

The best performing algorithm after the adaptation is the k-NN baseline, it reaches a MAE of 0.4206, a total improvement of 16.0% over the initial baseline benchmark.

Chapter 5

Discussion and perspectives

In the previous chapter, I described the goal and set-up of each experiment, and reported their findings. In this chapter, I summarize the findings and explain the learned lessons. I describe how these results can be used to provide better support and training to developers.

If nothing else, take away from this chapter...

The 2PL model has shown that some infamous vulnerability types that are typically considered high priority are relatively easy to find and fix in training. This is evidence of a gap between knowledge and practice. The paved path methodology, together with more usable developer tools, could help developers apply their knowledge better.
For use in the ITS, the k-NN baseline algorithm is the most accurate CF algorithm. Baselines have proven to be accurate for the data of the platform, most likely because many users show consistent bias in their ratings. This is related to the current item selection, that is often consistently right or consistently wrong, based on the ability level of the user. In literature, model-based algorithms are usually more accurate, but their advantages of dealing with data sparsity, scalability, synonyms, and implicit data are not applicable to the data set of the SCW training platform.
The results of the 2PL model and the ITS will be gradually implemented in the training platform. In the future the ITS will be improved to use data gathered from other developer tools, as described in a published patent by SCW.

5.1 Discussion

5.1.1 Two-parameter logistic model

Many efforts exist to rate, rank, organize, and prioritize vulnerabilities into lists and taxonomies, such as the OWASP Top 10 [93], the seven pernicious kingdoms [94], the Common Vulnerabilities and Exposures (CVE) [95–97], and the CWE/SANS Top 25 most dangerous software errors [98]. They are often built from the perspective of the security professional, and take into account prevalence in production, detectability by tools, and potential of impact. These lists and taxonomies can be used as guidelines for security professionals to decide which insecurities should be prioritized.

The results from the 2PL model sometimes contradict these lists. Injection and XXE, for example, are typically rated high on prioritization lists because of their prevalence in production software. According to our data, however, they are not the most difficult vulnerabilities in training. When the developer is aware such a vulnerability is present, they are able to detect and resolve it with relative ease. So while popular lists like OWASP Top 10 can be useful as baselines and guidelines for security teams to set their priorities, they might not be the right focus from an education perspective. Based on the results from the 2PL model, developer training should go further than these infamous security problems and focus more on security problems involving larger pieces of code. We can see this shift in priority in the newest version of the OWASP Top 10[1] as well. In the new version, shown in Figure 5.1, "Injection" goes down in priority and "XXE" even merges with "Security misconfiguration". New categories are introduced such as "insecure design", proposed as the new number 4, shifting the focus towards security flaws.

We see a clear gap between knowledge and practice with these vulnerabilities being so prevalent in practice, but relatively easy to fix in training. This is because, in a custom setting such as the SCW training platform, the developer is aware of security and able to apply their knowledge to the examples at hand. In practice, however, the developer is focused on the functionality and other requirements of the code, and security is no longer a priority. The cognitive burden to constantly keep track of both the functionality and the security of the code is evidently too large. In other research similar observations have been made, where half of the analyzed applications contained insecurities caused by oversights by the developer [79]. With the right processes and the right tools,

[1] `https://owasp.org/Top10/`

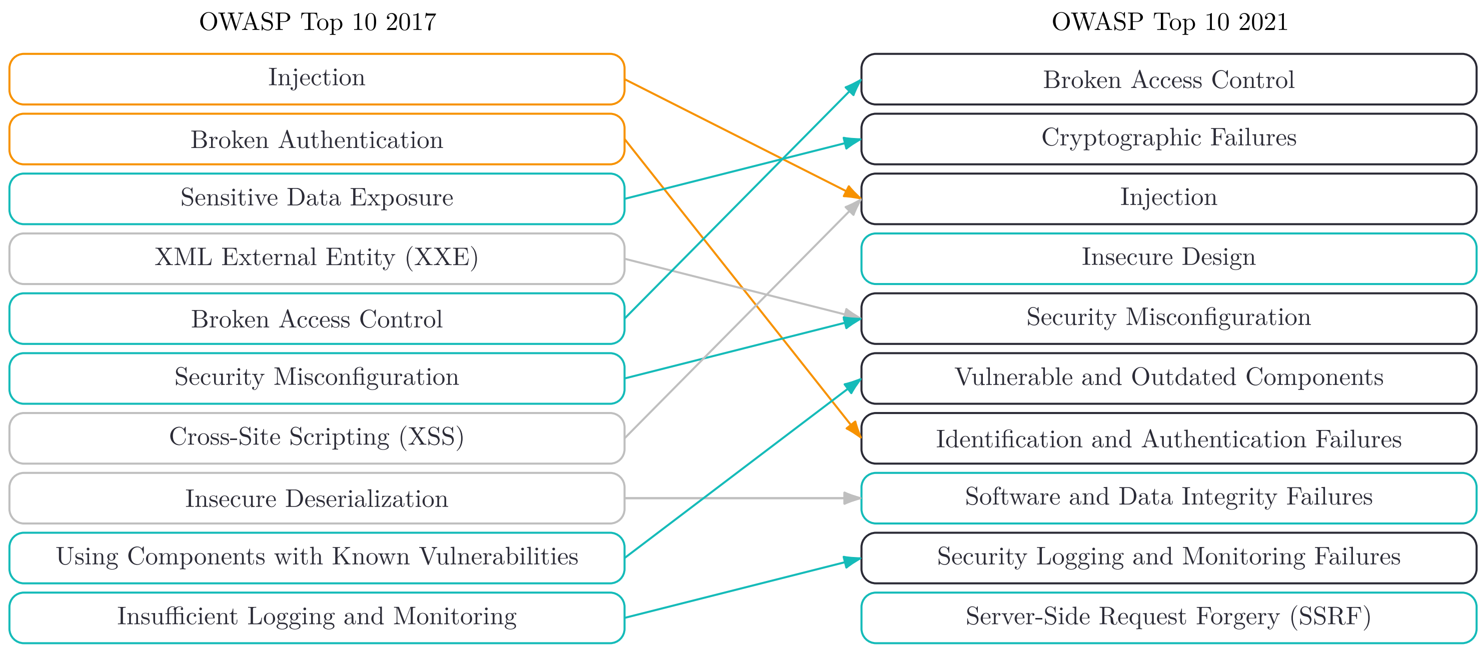

OWASP Top 10 2017
Injection
Broken Authentication
Sensitive Data Exposure
XML External Entity (XXE)
Broken Access Control
Security Misconfiguration
Cross-Site Scripting (XSS)
Insecure Deserialization
Using Components with Known Vulnerabilities
Insufficient Logging and Monitoring
OWASP Top 10 2021
Broken Access Control
Cryptographic Failures
Injection
Insecure Design
Security Misconfiguration
Vulnerable and Outdated Components
Identification and Authentication Failures
Software and Data Integrity Failures
Security Logging and Monitoring Failures
Server-Side Request Forgery (SSRF)

Figure 5.1: Some categories from the OWASP Top 10 2017 decrease in priority (marked in orange) in 2021, other categories increase in priority (marked in blue). Three categories are merged with other categories (marked in gray). In the 2021 version, three new categories are introduced (marked in blue).

this burden can be alleviated, and the prevalence of these vulnerabilities reduced. This is the goal of Part II of this work.

5.1.2 Recommendations

In the experiments described in the previous chapter, several algorithms were tested and adapted to learning systems. Memory-based algorithms based on baselines have come out on top. The best performing is the k-NN baseline algorithm using the baseline-centered Pearson similarity measure. This was expected based on the current exercise selection, as many users show a consistent bias in their ratings. Model-based algorithms based on dimensionality reduction through matrix factorization, are often the best performing algorithms. However, the advantages of these algorithms are often not applicable to our current data set.

Data sparsity In many recommender systems, the user-item rating matrix is rather sparse. In Netflix, for example, few users will have watched even half the catalogue of movies. This is also the case for the SCW training platform, especially for the largest frameworks that offer over a thousand challenges and by adapting the algorithms to learning systems, the sparsity of the data has even been increased artificially. This data sparsity can cause some problems for CF algorithms.

The *cold start problem* occurs when a new user or item enters the system. Because there is no item available about this user or item, it is difficult to find neighbouring items. Matrix factorization techniques reduce the dimensions of the matrix to alleviate this problem. In our data set only users have been included that completed a sufficient amount of challenges so that their ability level could be estimated, so this problem has been avoided. In practice, it will still be necessary to calibrate the ability level of new users before accurate recommendations can be made. One problem to avoid the cold start problem can then be to use a short entry test, in the form of CAT. A procedure like this is present in other learning systems, such as Duolingo.

For new items, a trade-off needs to be made between exploitation and exploration. In the exploitation phase the predictions from the algorithms are used to provide a recommendation to the user. In the exploration phase, an item is recommended for which there is insufficient data, risking a bad recommendation in order to gather new information about this item.

Scalability When the number of users and items is excessively large, computing the similarity between every two users becomes an expensive procedure. Model-based algorithms often scale better with large data sets because matrix factorization techniques are used for dimensionality reduction. While there are many users on the SCW training platform, and the number of users is only expected to grow, in practice the data sets are split per framework. They are not nearly large enough for scalability to be a problem, as similarity matrices for the k-NN algorithms are computed in less than a minute. These matrices only need to be computed once in a while, for example once or twice a day.

Synonyms Synonyms occur when a number of identical or similar items have different names or entries in the data set. Model-based techniques are capable of dealing with the synonym problem because they do not use the item names directly, but instead look for latent factors related to the items. In our data set we do not expect many natural synonyms to exist. While there are duplicate exercises, in the sense that they are in the same framework and about the same vulnerability type, in reality they are in different codebases, and of varying complexity.

With the learning adaptation, however, we have intentionally introduced synonyms by splitting items into separate entities based on the ability level of the users answering them. The fact that we still see significant improvements in the model-based algorithms, who are supposed to factor out item names, is proof that these items demonstrate significantly different characteristics in the latent factor space. This validates the hypothesis that user ability is an important factor for the recommendation of items in a learning system. It is possible that user ability is represented in the latent factor space in one way or another.

Shilling attacks In some recommendation systems (such as for example the Amazon web store) users can be compelled to give positive recommendations towards their own material and negative recommendations towards their competitors. While there is less incentive to do this type of intentional rating on the SCW training platform, similar scenarios have been detected. Users in one company made it a competition to see who could gather the most points, and they created bots for this purpose. The bots would randomly guess at first, and keep track of the correct answer for future attempts. This resulted in several users who answered all exercises in a single framework several times over, causing worse ratings for those items as these users did not learn anything new according to the IRT estimates. This has now been discouraged by preventing the same user from earning points through an exercise they have already solved in the past. For the experiments of this work, data generated by these bots has been filtered out.

Implicit data Implicit data has been briefly introduced in the explanation of the SVD++ algorithm. This algorithm not only characterizes users based on their ratings, but also based on which items they have rated. Using implicit data like this is expected to have a big impact on prediction accuracy for systems where the user can choose items themselves. In Netflix, for example, it can become apparent that a user constantly avoids watching movies of a certain genre, or that star a certain actor. On the SCW platform we also see an improvement, most likely because this allows the algorithm to better distinguish between users who follow the recommended courses, and those who do not.

5.1.3 Adaptation

The proposed adaptations to learning systems in this work are not specific to software security and could be applied to other domains. The adaptation based on processing of the data is especially easy to implement and can be applied to any CF algorithm. The biggest requirement is that sufficient data needs to be available to overcome problems caused by data scarcity which can be exaggerated by splitting the data even more. In learning systems more so than in movie or music recommendations, we can expect users to rate a significant portion of the items, which makes this requirement more likely to be met.

The adaptation was less effective in model-based CF algorithms. One possible explanation is that the latent factors from the dimensionality reduction techniques already represented an ability estimate. However, estimating this through the ratings alone is likely less accurate, which is why adding it more explicitly as a filter still improved the accuracy of the predictions. It is possible to imagine a similar adaptation in other contexts where the latent factors might be doing a good job already, but small improvements can be made by computing an important factor explicitly.

5.2 Perspectives

5.2.1 Implementation into the training platform

Results from the 2PL model can be used to improve the SCW training platform. This is a step by step process that has already started.

Quality control of the exercises The results of a 2PL model for the use in tests are used to remove items with a low discrimination parameter from the item bank. A low discrimination parameter means that this item it cannot differentiate well between users of high and low ability levels. Items like this are not useful in a test, where discriminating between users of different ability levels is exactly the goal. Before removing items with a low discriminative ability, the examiner often manually checks items that are only slightly below the predetermined threshold. If they are deemed important enough to be included in the tests despite their low discriminative ability, they are not removed from the item bank.

As a first step to use the results of the 2PL models in the SCW platform, we can use a similar procedure for quality control of the challenges. In contrast with tests, estimating ability is not the main goal of the ITS. Items with a low discrimination parameter might have little influence on the accuracy of ability estimates, but these items could still provide meaningful learning opportunities. A low discriminative ability can be explained by an extreme difficulty, or by a defect. If all users are answering the challenge correctly because it is extremely easy, or all users are answering it incorrectly because it is extremely hard, then this challenge can not be used to discriminate between users of high and low ability level. But if the discrimination parameter is low, and the difficulty level is not extreme, that means there is a different reason for this low correlation between a correct answer and the ability level of the user. A low discrimination parameter, in this case, is an indication that the challenge is misleading or defect. Challenges like this have been manually checked, and many of them have indeed shown defects in the past, or are still misleading in some way.

Improved challenge difficulty measure As explained in Appendix A, currently the difficulty of the challenges is only determined by the number of options to choose from. It is a probability of answering correctly in case of a blind guess. The results of the 2PL model experiment in Section 4.1 show that there is no statistically significant correlation between this difficulty and the probability of users answering the challenge correctly. It is not an accurate difficulty measure.

Nonetheless, this difficulty measure is used in tournaments and other gamification features to decide the amount of points that are awarded to a user after answering correctly. It would be more accurate to use the IRT difficulty for this purpose. This difficulty is only available for challenges for which there is a sufficient amount of data. There is still need for another measure to better approximate the real difficulty of new challenges that still have insufficient playtime.

In the experiment, I have shown that there is a correlation between the difficulty on one hand and the framework, the vulnerability type, and the presentation on the other hand. A good start for such a temporary approximation could then be to take the mean difficulty of all challenges in the same framework, about the same vulnerability, and with the same presentation.

Improved user ability measure Currently, a security maturity score is computed for each user based on the amount of points they have earned and the accuracy they have maintained while doing so. This maturity score is shown on the metrics dashboard, together with a more granular breakdown of the average strengths and weaknesses. This metrics dashboard is shown in Figure 2.2, on page 25. It is easy to reach a high maturity level for any user, if they spend enough time solving many easy challenges so that they earn points while maintaining a high accuracy.

The IRT ability estimate can be used to make this maturity score more meaningful. However, it currently only provides a global ability estimate and lacks the granularity required for the metrics dashboard. A multidimensional IRT model can be used to achieve this. It remains future work to train and evaluate such a multidimensional IRT model with, for example, one dimension for each vulnerability category on the platform.

Improved assessments The past few months, assessments have been the most used play mode on the platform. Assessments are built like classic tests, all users have to complete the same challenges and their accuracy is used as an indication of ability.

First, assessments can be improved by using IRT to estimate the ability level of the user. This ability estimate is more meaningful than the accuracy. This is most easily understood with an example: two users each complete an assessment with two challenges, one easy challenge and one difficult challenge. The first user makes a mistake (due to inattention) on the easy challenge, but answers the difficult challenge correctly. The second user answers the easy challenge correctly, but does not know the answer to the difficult challenge. These users have achieved the same accuracy on the assessment. However, intuitively, we would be more likely to attribute a higher ability level to the first user. This is exactly what can be achieved when IRT is used.

Second, assessments can be made adaptive, similar to a CAT. This means, serving new challenges based on the temporary ability estimate during the assessment. Not only will this improve accuracy of the ability estimate, it will also reduce the amount of challenges needed to complete an assessment.

Recommendations in training Once all other implementation related to the 2PL model have been completed, the CF algorithm can be integrated in the platform to dynamically recommend challenges to each user in the training mode.

5.2.2 Integrating with developer tools

SCW is currently developing integrations for several developer and security tools such as Jira[2], GitHub[3], Fortify[4], and more[5]. Additionally, there is also an integration for the Integrated Development Environment (IDE), discussed in great detail in Part II of this work.

The current goal of these integrations is to provide contextual training to developers. When a security vulnerability is detected through Fortify, or a ticket that involves a vulnerability is created in one of the bug tracking systems, the integration will insert links to training on the SCW platform about this specific vulnerability. This training is highly relevant since it is directly related to the task at hand, i.e. remediating the vulnerability.

However, it is my opinion that they might hurt the productivity and usability of the developer. I believe the developer does not want to make a context switch to complete training exercises, when they should be resolving the problem. It would be more beneficial to let these integrations insert project-specific remediation guidance, as will be explained in Part II.

[2] `https://marketplace.atlassian.com/apps/1221320/`

[3] `https://github.com/marketplace/secure-code-warrior-for-github`

[4] `https://www.microfocus.com/en-us/fortify-integrations`

[5] `https://help.securecodewarrior.com`

Most of the integrations are still in development, and are only used by a select few customers. In the future, when more data is available, it will be interesting to analyse which of these inserted links are clicked most frequently. This data can give an indication for which vulnerabilities developers seek out training, and hence which vulnerabilities are likely more difficult to understand and solve in practice. It would be interesting to compare these results to those of the 2PL model and priority lists such as the OWASP Top 10.

Adaptive Security Guidance Instead of forcing a developer to make a context switch to follow this contextual training, I have invented an alternative solution. With this system, contextual training can be provided at a more appropriate time, that is, when the developer opens the training platform. This invention has been patented by SCW as a "Method and System for Adaptive Security Guidance"[6].

In this invention, the data collected from the integrations feeds into the ITS to result in even more relevant, and highly applicable recommendations. For example, the IDE integration allows us to monitor code changes the developer makes and verify them against a set of rules to detect the mistakes they make in practice. Integrations with security scanners such as Fortify in combination with code repository tools such as GitHub, also allow us to collect data about the detected vulnerabilities and who is responsible for those pieces of code. At the same time, integrations with issue tracking systems such as Jira allow us to detect which tasks a developer is assigned to, and whether there are any security problems or security-critical features among them. All of this data, in combination with the performance of users on the platform itself, enables us to make highly relevant recommendations.

Implementing such a system is most likely achieved by first selecting a list of relevant items to recommend and then choosing the most appropriate. If a user is assigned a ticket on Jira regarding a SQL injection, for example, a challenge needs to be selected from the list of challenges about SQL injection in the relevant language and framework. Out of this list, the challenge with the highest predicted rating can be recommended to the user. Implementing and evaluating this system remains future work.

[6]`https://patents.google.com/patent/US20200211135A1/en`

5.2.3 Mobile application

During my research at SCW I have been closely involved in the design of a mobile application called Secure Code Bootcamp[7]. Besides videos and texts explaining different vulnerability types, developers can also play challenges similar to those on the training platform. The challenges for the mobile app can be generated from the same vulnerability data as those on the platform.

Instead of presenting it as an identify, locate, or fix exercise, a new presentation form has been developed that is more suitable for a small screen. In these challenges the user has to review a code sample and either accept or reject it based on the security of the sample. To accept or reject, the user can tap a button, or swipe to the left or right in a Tinder-style interaction with the app.

The app is used by a few hundred students and developers but has not yet had as much use as the SCW platform itself. In the future, analyzing the learning behaviour of the users on this app can provide further insights in the effect of different types of exercises, and the mobile context of the education.

[7] `https://www.securecodewarrior.com/products/secure-code-bootcamp`

Part II
Tools

Do, or do not. There is no try.

Yoda

Introduction

Despite growing efforts to educate developers, they still frequently make mistakes in practice. Because security experts are understaffed and unable to assist each developer, security tools have become part of the developer's arsenal.

However, security tools are not designed with the developer in mind and are often a big inhibitor of their productivity. As a result, developers dislike and often disable these security tools.

Engineers at SCW designed and implemented an IDE plugin called Sensei. I evaluated early prototypes and helped redesign and shape Sensei to the tool it is today. Sensei is a tool in line with the paved path methodology as it has a heavy focus on developer usability and productivity. It checks compliance of code being written to a set of coding guidelines, similar to an as-you-type spell checker. Upon detected violations, it offers remediation guidance and additional information to the developer. Sensei allows developers and security experts to develop customized rule sets to be enforced, specific to each project.

In this part, I first describe the evolution of traditional security tools in Chapter 6. I explain where they are lacking as a tool to support development in the paved path methodology and what goals and requirements to set instead. In Chapter 7, I describe the Sensei IDE plugin, its design, and its features. I continue by describing the experiments and observations of the tool in Chapter 8. In Chapter 9, I discuss the findings of the experiments and the lessons learned and also offer some perspectives that remain future work.

Chapter 6

Goals and requirements

Traditional security tools are an important part of the SDLC, and will remain so in the foreseeable future. However, they hinder productivity and do not integrate well in the workflows of developers. To provide developers with more suitable tools, it is critical to understand the goals and the shortcomings of traditional security tools. In this chapter, I briefly describe how security practices and tools evolved over time and explain the disconnect with developer workflows. I also describe how the paved path methodology can improve this situation.

If nothing else, take away from this chapter...

The goal of traditional security tools is automation of security testing. In order to keep up with the ever increasing speed of software development, they are *shifting left* in the SDLC, towards the development phase. As a result they are being integrated in developer tools. However, they are still fundamentally using a reactive approach, scanning (partly) completed code and its calling context for vulnerabilities. To fix detected vulnerabilities, developers often have to go back to the code (potentially long) after it was initially developed. With tools in the paved path methodology, the goal is a preventative approach. Guidelines are enforced regardless of context as the code is being written. This helps the developer write secure code from the start, improving their productivity. As a result, code fragments are being secured, even if no current calling context exists that leads to an exploit. The fragments are being secured for future use.

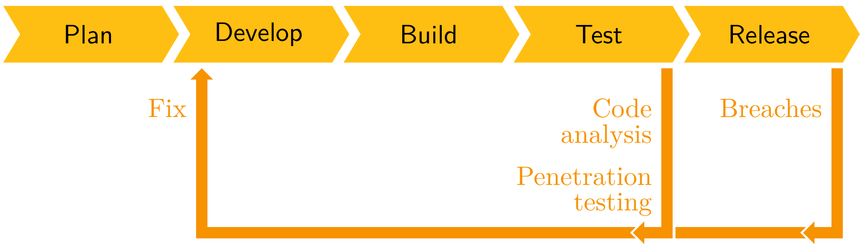


Figure 6.1: Historically, security was considered a part of software testing and addressed from the end (right) of the SDLC.

6.1 Traditional security tools

For a long time, security has been considered a part of software testing [99]. Security was addressed in a reactive manner, from the end (right) of the SDLC, as shown in Figure 6.1. Based on vulnerabilities reported when the application was tested after its initial development was completed or when it was already deployed and in production, developer training was adapted, new coding checks were introduced, and security problems were fixed by revisiting code.

Experience has shown, however, that security should not be an afterthought of software development but that it should be addressed earlier in the development. This is not only to minimize costs [100–103]. Shorter feedback loops also result in better learning performance [104, 105]. As a result a *shift left* movement is ongoing to try to identify possible security problems as early as possible in the SDLC, as illustrated in Figure 6.2. New project management techniques such as Agile and DevOps encourage fast incremental releases where the developer is also responsible for meeting non-functional requirements such as security. To support that, testing and deployment of security guidelines needs to be more automated in short feedback loops, thus shifting security left.

While supporting the shift left, conventional vulnerability scanning tools still use a reactive, testing-based approach. Furthermore, in training developers are typically taught how certain mistakes lead to vulnerabilities, and how these can be exploited. Afterwards they are taught how to prevent the presented vulnerabilities. These practices are extended into the development phase, where the focus is still on the (sometimes complex) question of whether or not the code is vulnerable, and only if it is considered to be vulnerable, it becomes a candidate to be fixed. The shift left movement is certainly an improvement, but it is not yet perfect. Many security problems still occur. Companies acknowledge

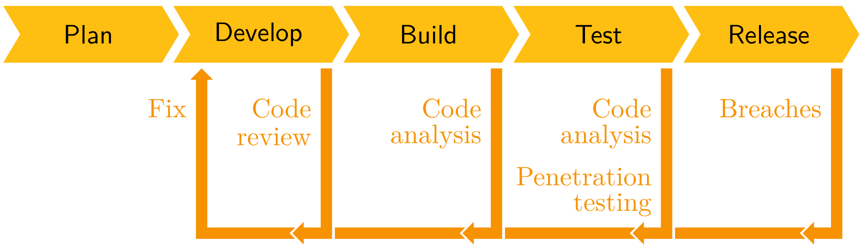


Figure 6.2: In the shift left movement, security practices are shifting left in the SDLC. This results in shorter feedback loops, but is still using a reactive approach to find problems after they have been introduced.

this, as is obvious from the incentives they put in place to minimize the impact of potential breaches, such as bug bounty programs.

Many of the vulnerability scanning tools use complex control flow and data flow analyses to scan for vulnerabilities in the product. They identify, e.g., user input that is not properly validated and passed on to security-critical parts of the application. If it is determined that a malicious input exists that can cause unwanted or unexpected results, these issues are placed into the bug tracking system for developers to deal with. In order to successfully detect vulnerabilities, the calling context of routines needs to be known in order to perform the necessary global analyses. Because of this, such tools can only be deployed at a later stage in development. It is, in other words, not possible to shift even more to the left with only these tools. During the earlier development stages of a product, it is entirely possible that no user input can reach a buggy routine yet. The classic approach will only flag the routine once the context exists where it can be exploited. This then requires the developers to go back to secure the routine at a later time than when they were originally developing it.

In short, even in the ongoing shift left movement, the problem is still approached from the right side of the SDLC, following the detection of vulnerabilities. The detection is shifted as much to the left as possible but the approach is still reactive, and requires revisiting code (possibly long) after it has been developed.

6.2 Tools for the paved path methodology

In the paved path methodology we try to avoid this reactive approach. Instead, the goal is to prevent the introduction of security problems as much as possible, as shown in Figure 6.3. This is achieved by laying

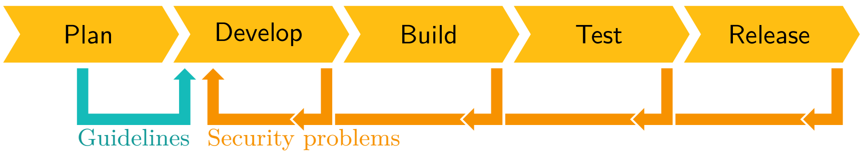


Figure 6.3: The paved path methodology introduces a preventative approach. This is done by creating guidelines that, when adhered to, will help prevent the introduction of security problems.

out guidelines early in the process and enforcing them regardless of the calling context of the code. In code that does not take user input, and hence is not likely to result in vulnerabilities, the guidelines are still enforced. It is after all possible that in the future a calling context will be developed that does take user input. The code may become vulnerable at that point. Securing this code fragment from the start will protect it for future use and avoids the need to revisit and fix it when such a calling context exists. This practice is often called "establishing secure defaults", and it is part of a "security by design" approach in software engineering[1]

This fundamentally different approach of enforcing (secure) coding guidelines instead of scanning for vulnerabilities, makes it possible to meet the requirements for tools supporting the paved path methodology, such as Sensei. Sensei is an IDE plugin developed by SCW with the goal of helping developers produce more secure code. It is currently available for IntelliJ IDEA and Android Studio, it supports Java, Kotlin and Extensible Markup Language (XML). Sensei can be used by DevSecOps teams to apply the paved path methodology in their software development process. As described in Chapter 1, to support the paved path methodology, Sensei needs to be *relevant*, *efficient*, and *usable*.

In order to be *relevant* to the developer's work, the paved path methodology prescribes to create API-level guidelines that determine which libraries and even which library calls are to be used in the project. Custom (wrapper) libraries may need to be developed that are inherently safe so they can be freely used by developers. To meet this requirement, the guidelines enforced by Sensei need to be easy to customize. For this purpose Sensei offers a custom editor inside the IDE which will be discussed in more detail in Chapter 7. Easy customization of the guidelines enables security experts and developers to efficiently create and enforce new guidelines as a way to share their knowledge among the rest of the

[1] `https://wiki.owasp.org/index.php/Security_by_Design_Principles`

team.

Sensei is designed to be a developer tool in the first place. It is *efficient* as it improves developer productivity instead of hurting it. Sensei enforces coding guidelines regardless of context. Since the context can be ignored, it only needs to perform local code analyses that can be completed in real time, similar to an as-you-type spell checker. Also similar to a spell checker, Sensei provides an easy way to remediate guideline violations in the form of quick-fixes. These code transformations are an existing IDE feature that the developer is familiar with. They are commonly used for marking syntax errors and general coding best practices. With quick-fixes it is possible to avoid the need for research and even automate the remediation of guideline violations, which greatly improves the productivity of the developer.

Quick-fixes turn the task of fixing insecure code into one where the developer has to *recognize* the correct solution, rather than *recollect*, reducing the cognitive effort and improving *usability*. Sensei and its quick-fixes are also used by developers for other purposes than security, as will be discussed in the next chapters. Because the tool resides in the IDE and reuses existing IDE features it quickly feels like a simple extension of the existing developer tool kit.

Chapter 7

Sensei

The development of the Sensei IDE plugin started in 2016, when dr. Matias Madou and Nathan Desmet founded the company Sensei Security. I joined this company, that later would merge with SCW, as an intern a few months later. When I started my research in 2017, I set forth to discover how this tool could be used most effectively, to evaluate its concepts and features, and to help direct its design. In this chapter, I describe the Sensei IDE plugin and discuss the lessons we learned during the implementation and testing of the tool.

If nothing else, take away from this chapter...

The first iteration of the Sensei rule editor was a Graphical User Interface (GUI) containing many input fields to allow fast customization of rules. It was used by us to create hundreds of rules for customers and developer communities which frequently resulted in the need for extra features. Some of these features are useful for improving the context awareness of Sensei and its usability, other features fell out of use. Eventually, through the addition of these many features, the rule editor became too cluttered and unclear.
As a more flexible alternative, Nathan Desmet, principal engineer at SCW, and I designed a new formatting language based on YAML Ain't Markup Language (YAML) syntax that allows rule-writers to quickly and effectively create rules and quick-fixes. The rules include several features to improve their usability and add support for libraries and design flaws.

7.1 Recipes

The API-level rules that are enforced by Sensei are called recipes. This name is chosen to emphasize the difference between Sensei and existing, traditional security tools, which often use rules to scan for vulnerabilities. Recipes are also commonly used in the DevOps movement, for example by the popular automation tool Chef [1].

7.1.1 Creating recipes

Customization and distribution of the recipes is a crucial feature for any successful tool supporting the paved path methodology. If the recipes are easy to customize, Sensei can be more easily tuned to provide highly relevant and applicable feedback to the developer. This customization should be scalable and hence not be a service provided by engineers or experts at SCW. It should allow developers and security experts to effortlessly share project-specific or team-specific guidelines among each other. For this reason, the recipe creation process should be easy and fast, and at the same time versatile.

Our first approach allowed users to create new recipes through predefined recipe models. A GUI was used to let the recipe-writer fill in a number of variables for this model. A simple example of such a model is the "Replace method call model". Figure 7.1 shows a recipe being created to replace the `addCookie` method with a safe alternative from the OWASP Enterprise Security API (ESAPI), an open-source, web application security control library designed to make secure development easier [106]; the organization also provides some commonly used security guidelines. The recipe-writer has to fill in some specifics about the method they want to be marked by Sensei, such as the package, class, and method names. Then they can write one or more quick-fixes. To create a quick-fix, they have to write a quick-fix description and they have to define the code fragment that will be used to replace the original. For the replacement code, they can reuse arguments, method names, and more from the original code by means of a template language. In the field "Rewrite to", the example quick-fix reuses the first argument of the original method call by using the template `arguments.0`.

However, for more complex models the number of input fields grew rapidly to accommodate a plethora of corner cases, and so did the number of models for multiple scenarios. As of now the old recipe editor has over 40 different models. With this many models, it becomes overwhelm-

[1] `https://www.chef.io/`

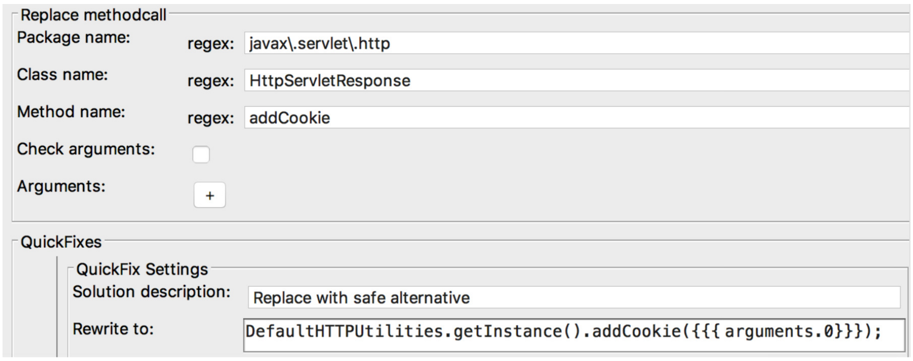

Replace methodcall
Package name: regex: javax\.servlet\.http
Class name: regex: HttpServletResponse
Method name: regex: addCookie
Check arguments:
Arguments:
QuickFixes
QuickFix Settings
Solution description: Replace with safe alternative
Rewrite to:
```
DefaultHTTPUtilities.getInstance().addCookie({{{ arguments.0}}});
```

Figure 7.1: The old recipe editor used a GUI. It required the recipe-writer to fill in a number of input fields to specify the behaviour of the recipe.

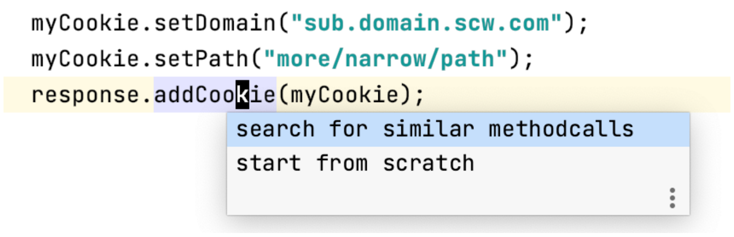
ing for recipe-writers that have to select a model to enforce their desired coding guideline. The described model-based recipe creation process is not flexible and intuitive enough, so in the next iteration Nathan Desmet and I designed a new approach.

In this approach, we split up the recipe in two parts: A trigger to identify the violation, plus an optional quick-fix to correct the vulnerability consistently according to company best practices. Triggers are now specified by way of YAML[2] syntax, which provides more flexibility. To develop this YAML syntax, all existing Sensei rules were analyzed and grouped based on which elements in the code are incorrect and which transformations are required to fix them. The resulting taxonomy of 10 bad code patterns is included in Appendix D.

Since this approach requires recipe-writers to learn the new syntax, we have provided some tools to assist them, in the form of a GUI that can be used to build the desired recipe from scratch. In addition, the recipe editor is now more context-aware. The recipe-writer can open a recipe creation wizard by pressing a key combination in the text editor in the IDE and selecting "create new recipe". This opens a context-aware menu depending on the position of the caret. For example, if the caret was on a method call, the menu contains an option to create a new recipe that searches for similar method calls, as shown in Figure 7.2.

When this context-aware option is chosen, the recipe creation wizard is opened and a recipe is automatically suggested from the available context. To search for a methodcall, the information that can be pre-

[2] https://yaml.org/



```
myCookie.setDomain("sub.domain.scw.com");
myCookie.setPath("more/narrow/path");
response.addCookie(myCookie);
```

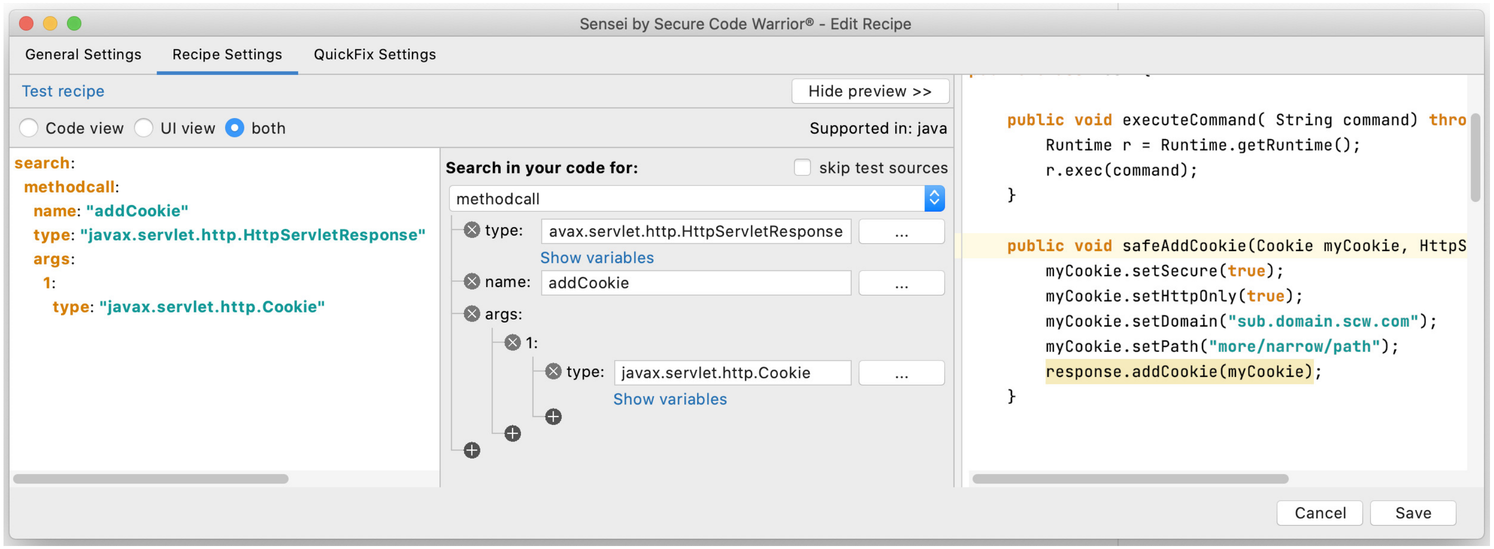
Figure 7.2: The recipe creation menu is context aware, its options will change based on the caret position.

filled from context is the type and the name of the methodcall, as well as the number of arguments and each of their types. The user can then adjust the recipe to reach the desired results through the YAML code or the provided GUI. This window also provides a preview panel, as shown in Figure 7.3. In this panel, the code file from which the recipe wizard was opened is shown, and the effects of the recipe being created are visualised, which allows for easy customization.

After creating a trigger, it is possible to create an optional quick-fix. Here, the recipe-writer has to fill in the quick-fix description and the replacement code. For the replacement code, they can make use of the same template language as in the first approach to reuse parts of the original code. Below the input field, an overview is provided of the available parts of the original code, as shown in Figure 7.4. Double clicking one of these options, adds its template to the fix. The quick-fix creation window also offers a live preview in the lower right corner that highlights the changes that would be made to the original code (shown in the lower left corner) if the quick-fix is applied.

Finally, besides the trigger and the fix, there are also a number of general settings for the recipe that can be configured, as shown in Figure 7.5. Some examples are the name, descriptions, the category of a related vulnerability, overriding recipes, and scopes. All of these features are related to the usability of the developer and will be discussed in the following sections.

When creating recipes in-house, we have observed that the context-aware recipe wizard has greatly sped up the recipe creation process. In practice, creating new recipes often starts from a bad code example, either when fixing a vulnerability or while reviewing the code of a colleague. The recipe-writer can then simply open the recipe creation wizard from this example. The live previews also greatly improve the usability, since before they were introduced, to create a finished recipe the recipe-writer



Code view UI view [✓ both]   Supported in: java

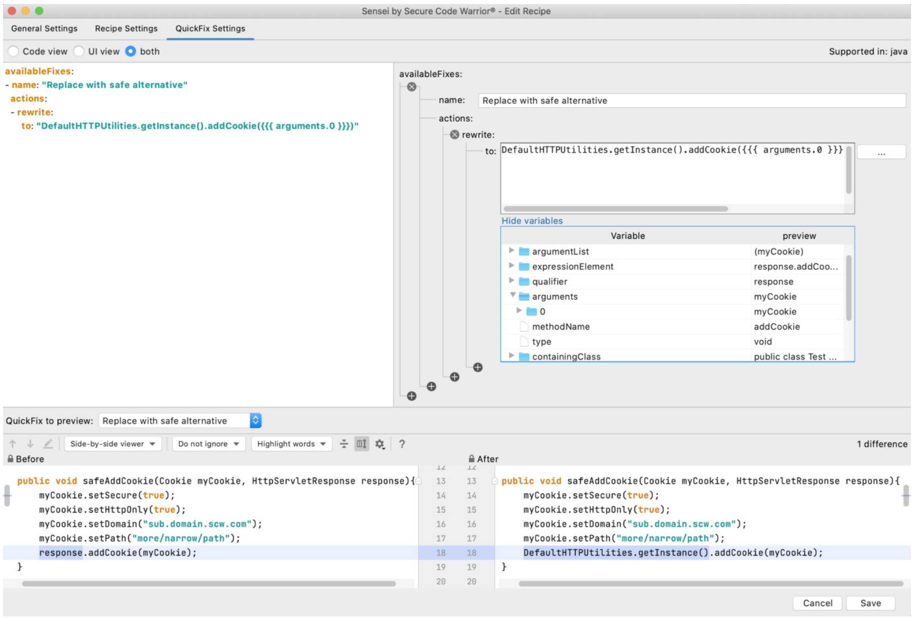
```yaml
search:
  methodcall:
    name: "addCookie"
    type: "javax.servlet.http.HttpServletResponse"
    args:
      1:
        type: "javax.servlet.http.Cookie"
```

Search in your code for: skip test sources

methodcall
- type: avax.servlet.http.HttpServletResponse ...
  Show variables
- name: addCookie ...
- args:
  - 1:
    - type: javax.servlet.http.Cookie ...
      Show variables

```java
    public void executeCommand( String command) thro
        Runtime r = Runtime.getRuntime();
        r.exec(command);
    }

    public void safeAddCookie(Cookie myCookie, HttpS
        myCookie.setSecure(true);
        myCookie.setHttpOnly(true);
        myCookie.setDomain("sub.domain.scw.com");
        myCookie.setPath("more/narrow/path");
        response.addCookie(myCookie);
    }
```

Figure 7.3: A recipe created through the "search for similar methodcalls" option in the context-aware recipe creation menu will generate a YAML-based recipe with details from the context of the caret position.


Sensei by Secure Code Warrior® - Edit Recipe

General Settings | Recipe Settings | QuickFix Settings

Code view ( ) UI view ( ) both (●)   Supported in: java

```
availableFixes:
- name: "Replace with safe alternative"
  actions:
  - rewrite:
      to: "DefaultHTTPUtilities.getInstance().addCookie({{{ arguments.0 }}})"
```

availableFixes:
name: Replace with safe alternative
actions:
rewrite:
to: DefaultHTTPUtilities.getInstance().addCookie({{{ arguments.0 }}})

Hide variables

Variable	preview
argumentList	(myCookie)
expressionElement	response.addCoo...
qualifier	response
arguments	myCookie
0	myCookie
methodName	addCookie
type	void
containingClass	public class Test ...

QuickFix to preview: Replace with safe alternative

Side-by-side viewer | Do not ignore | Highlight words | 1 difference

Before

```
public void safeAddCookie(Cookie myCookie, HttpServletResponse response){
    myCookie.setSecure(true);
    myCookie.setHttpOnly(true);
    myCookie.setDomain("sub.domain.scw.com");
    myCookie.setPath("more/narrow/path");
    response.addCookie(myCookie);
}
```

After

```
public void safeAddCookie(Cookie myCookie, HttpServletResponse response){
    myCookie.setSecure(true);
    myCookie.setHttpOnly(true);
    myCookie.setDomain("sub.domain.scw.com");
    myCookie.setPath("more/narrow/path");
    DefaultHTTPUtilities.getInstance().addCookie(myCookie);
}
```

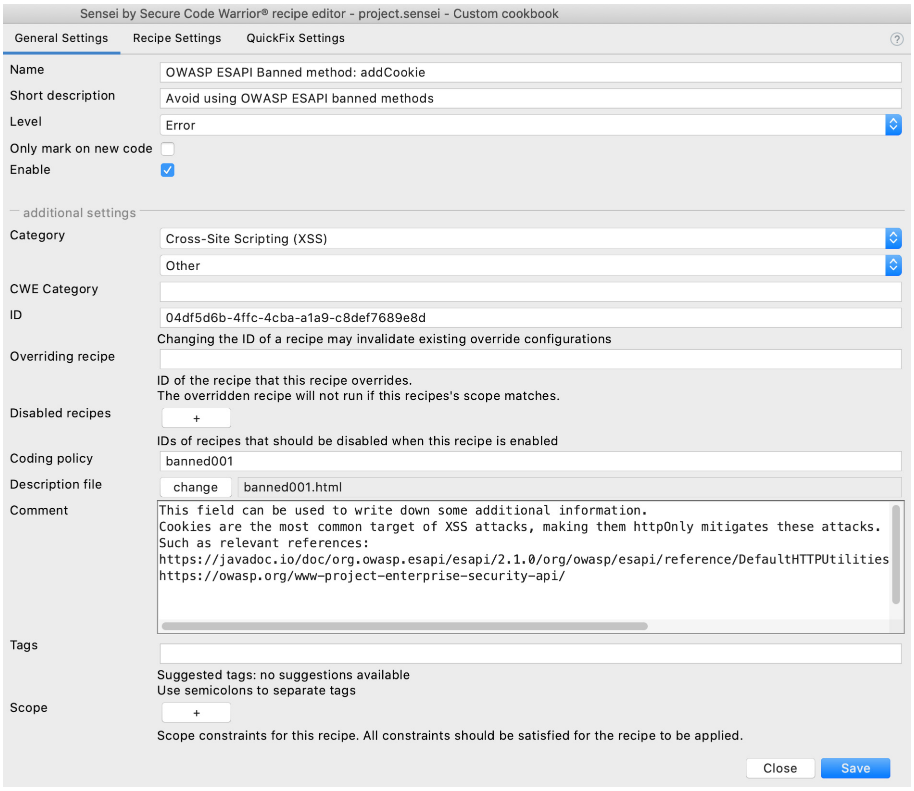
Figure 7.4: The fix creation window allows the recipe-writer to reuse parts of the original code.

was required to go back and forth several times to test the recipe in the IDE and adjust it in the recipe editor.

7.1.2 Managing recipes

In the paved path methodology, guidelines can be put in place at the start of the project. If not, at the very least, relevant guidelines should be created each time a new feature is going to be developed. Together with those guidelines, Sensei recipes should be created as well. The recipes, however, can also be used by the developers themselves, as a way to share knowledge. When they develop new APIs, additionally to documentation, developers can also add Sensei recipes to the project that help their colleagues use these APIs as intended.

We also recommend to make Sensei recipes part of the remediation process when problems are found by security testing or reported through bug bounty programs. It should be part of the process to create a recipe that prevents this same vulnerability from occurring in the future. Currently, it is often the case that security experts run the security scans. When problems are found, these experts guide the developers by providing them with informal, broadly applicable guidelines and checklists.



Name	OWASP ESAPI Banned method: addCookie
Short description	Avoid using OWASP ESAPI banned methods
Level	Error
Only mark on new code	☐
Enable	✓
Category	Cross-Site Scripting (XSS)
	Other
CWE Category	
ID	04df5d6b-4ffc-4cba-a1a9-c8def7689e8d
	Changing the ID of a recipe may invalidate existing override configurations
Overriding recipe	
	ID of the recipe that this recipe overrides. The overridden recipe will not run if this recipes's scope matches.
Disabled recipes	+
	IDs of recipes that should be disabled when this recipe is enabled
Coding policy	banned001
Description file	banned001.html
Comment	(see below)
Tags	
	Suggested tags: no suggestions available Use semicolons to separate tags
Scope	+
	Scope constraints for this recipe. All constraints should be satisfied for the recipe to be applied.

Comment:

```
This field can be used to write down some additional information.
Cookies are the most common target of XSS attacks, making them httpOnly mitigates these attacks.
Such as relevant references:
https://javadoc.io/doc/org.owasp.esapi/esapi/2.1.0/org/owasp/esapi/reference/DefaultHTTPUtilities
https://owasp.org/www-project-enterprise-security-api/
```

Figure 7.5: Some additional settings are available in the recipe editor mostly related to usability of the developer.

These instructions sometimes use security jargon that might not be clear to all developers, and even if they are understood, that does not guarantee the developer will be able to apply them in practice. In the paved path methodology, security experts and developers should work together to create API-level guidelines instead. As part of this process, to communicate these guidelines to the rest of the team, Sensei recipes can be created as well.

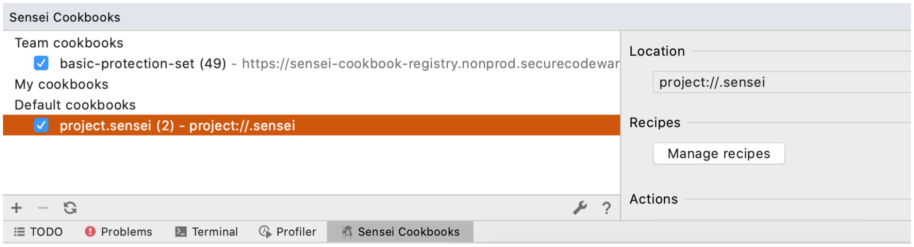
For existing projects, we recommend companies to start with no recipes and use existing data on the security of their project as a starting point. This could be the report of a penetration test, or results of vulnerability scans. While resolving these issues in the code, developers and security experts can start building the first recipes. Some clients of SCW have been hesitant to start with an empty security tool and, despite our recommendations to customize recipes for each project, still wish to receive starting recipes. For this reason we have created small open-source sets of unopinionated recipes that can be used in all projects[3]. These recipes aid in correctly using the standard libraries of certain popular frameworks (e.g., Java EE, Android Software Development Kit (SDK), Amazon Web Services (AWS) SDK). This set can be used as inspiration and to get both security experts and developers accustomed to the tool, but usually it does not flag many issues.

Considering the different sources of recipes, developers can have recipes imposed by management and/or by the security team, as well as recipes distributed among the developers per team or project. On top, there is the open-source recipes that can be used as a starting point when first using the tool. To make the management of recipes easier, we group recipes into cookbooks. Instead of distributing recipes one-by-one, this allows for grouping and distributing related recipes more easily.

In order to manage these cookbooks in the IDE, a cookbook manager is provided, as shown in Figure 7.6. Each cookbook is specified by a name and a location. The developer can enable or disable any cookbook as well as edit recipes in some cookbooks.

Cookbooks can be stored locally or remotely. Remote cookbooks are called team cookbooks and can be loaded from a github project (e.g., `git@gitserver:cookbooks|master|recipes`) or another remote server location (e.g., `https://remote.com/recipes.zip`). Remote cookbooks are only recommended to distribute generally applicable cookbooks, since remote recipes are not editable by the developers and are instead read-only. Any updates to the remote cookbooks are automati-

[3] `https://securecodewarrior.github.io/public-cookbooks/`


Sensei Cookbooks

Team cookbooks
- ✓ basic-protection-set (49) - https://sensei-cookbook-registry.nonprod.securecodewar

My cookbooks

Default cookbooks
- ✓ project.sensei (2) - project://.sensei

Location: project://.sensei

Recipes: Manage recipes

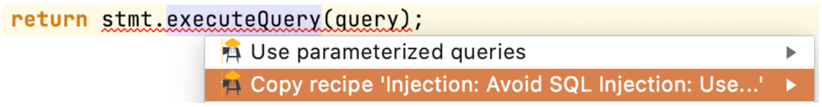
Actions

Figure 7.6: The cookbook manager in this screenshot contains one remote team cookbook as well as one default cookbook stored in the project structure.

cally pushed to all the developers. Locally stored cookbooks are editable and can be specified by a local path (e.g., `/Users/dev/recipes`) for personal cookbooks or a path starting from the project root (by default `project://.sensei`) for default cookbooks for a project. Local cookbooks are editable which means they can also be enabled on a recipe-by-recipe basis. It is advised to store project-specific recipes as part of the project. This way, the recipes are always available, up-to-date with the code, and following the same flow as regular code (e.g., branch, review, merge). When recipes follow the same flow as the code, new APIs and the recipes needed to use them properly can be added to the project and reviewed as a whole.

The paved path methodology encourages customization of the recipes at project level. Previous research and experience have shown that customization at this level is the most successful. This provides the needed flexibility to tailor the enforced coding guidelines to the code, but also ensures that the team has a joint approach to how the code for a project should be developed [107]. Individually customized recipes might lead to disagreements, while company-wide recipes might be too general to be easily applied.

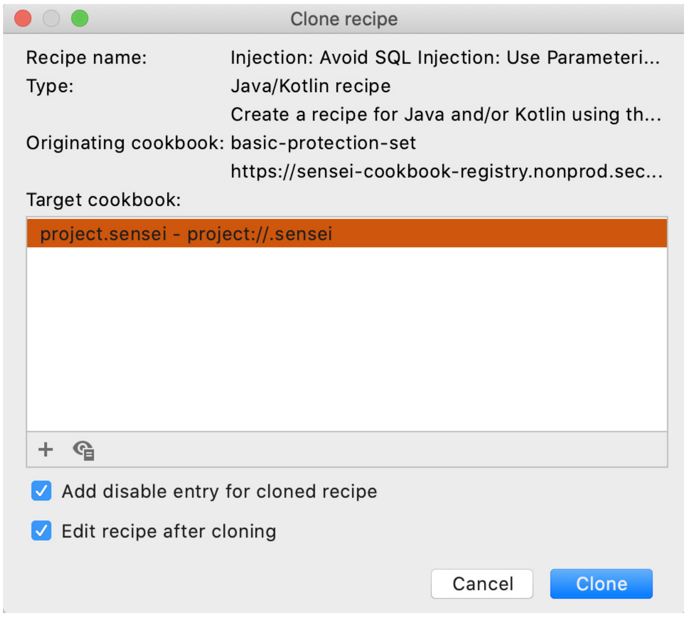
It is possible for a recipe to be configured to disable other recipes, as shown in the general settings in Figure 7.5. This feature can be used to improve remote, read-only recipes. It is possible that such a recipe is not fully applicable to the project, e.g., because it requires too many manual adaptations. It is then possible to create a replacement recipe that can be distributed to one team or project and disables the original recipe when it is active. To facilitate this, an option in the quick-fix menu is added to copy remote recipes to a local cookbook, as shown in Figure 7.7. This option can easily be hidden in the settings. The clone recipe window, shown in Figure 7.8, provides an option to automatically



```
return stmt.executeQuery(query);
```

Use parameterized queries
Copy recipe 'Injection: Avoid SQL Injection: Use...'

Figure 7.7: For remote recipes, the quick-fix menu offers a "Copy recipe" option.



Recipe name:	Injection: Avoid SQL Injection: Use Parameteri...
Type:	Java/Kotlin recipe
	Create a recipe for Java and/or Kotlin using th...
Originating cookbook:	basic-protection-set
	https://sensei-cookbook-registry.nonprod.sec...

Target cookbook:

project.sensei - project://.sensei

☑ Add disable entry for cloned recipe

☑ Edit recipe after cloning

Figure 7.8: The clone recipe window allows the recipe-writer to configure the new recipe to disable the recipe it is copied from.

disable the original recipe it is copied from. When a remote recipe is disabled or replaced, the author of this recipe should be notified. It is possible that it is a generally applicable improvement and the recipe can be updated accordingly for other teams or projects that use it in a remote cookbook. An additional quick-fix option will be added in the future to disable a recipe.

The discussed features to disable recipes have been designed to improve the usability of the tool for developers. They are in line with the philosophy that developers' productivity benefits from their ability to customize their development environment to their preferences, and to give them a significant amount of freedom in that regard. In that philosophy, it is preferable to have developers disable some recipes rather then uninstalling or neglecting the tool completely. Importantly, this does not necessarily result in guideline violations slipping below the radar, since security and management can still have these recipes, as well as complimentary tools, enabled in later phases of the SDLC.

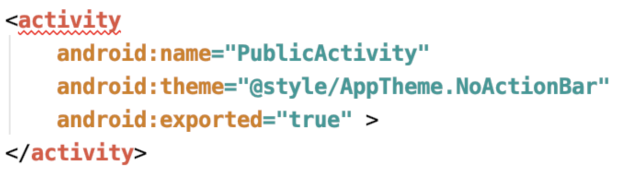
7.1.3 Verifying recipes

To inspect the code against a number of recipes, our tool reuses the IDE syntax checking features. When a developer writes new code, the IDE rebuilds the Abstract Syntax Tree (AST) and computes the changes compared to the previous version. A limited AST of the changes, containing the necessary symbol information, is then passed on, allowing tools to only analyze the changes. On this AST, a combination of specialized light-weight versions of existing analysis techniques is used such as taint analysis, data flow analysis, and control flow analysis to verify the recipes in real time.

7.1.4 Explaining recipes

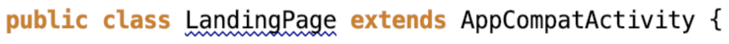
In order to mark violated guidelines, our plugin makes use of existing IDE features to flag coding mistakes. In most IDEs the code markings by default have three levels of severity: *error*, *warning*, and *information*. We recommend to mark coding guideline violations as errors. Traditional error-level markings are usually immediately addressed by the developer, while warning-level markings are more frequently ignored [105]. This is the case because error-level warnings in an IDE typically indicate a problem in the code that will result in a compilation failure. Currently error markings by our tool still allow successful compilation of a project, but several clients have requested for the markings to result in compilation failures, equivalent to errors marked by the IDE itself. This is not surprising, as it is in line with the default behavior of popular IDEs such as Visual Studio. For example, when Visual Studio's C compiler compiles code that uses the insecure `sprintf` function, it throws a compilation error warning the developer that the function may be unsafe.

An example marking can be seen in Figure 7.9, where the opening `<activity>` tag in XML code is marked as an error. This marking makes the code fragment stand out and attracts the developer's attention. In the example, the Android activity is configured as a public activity by setting the `exported` attribute to `true` but not configuring an intent filter. In XML code, like in this example, it is advised to only mark the opening tag, and not the entire XML tag and its content. This would overwhelm the developer, and it would not be clear which part of the code is lacking.



```
<activity
    android:name="PublicActivity"
    android:theme="@style/AppTheme.NoActionBar"
    android:exported="true" >
</activity>
```

Figure 7.9: XML recipes can be configured to mark the opening tag only (shown in the figure), the opening tag and the closing tag, or both tags and their entire content.



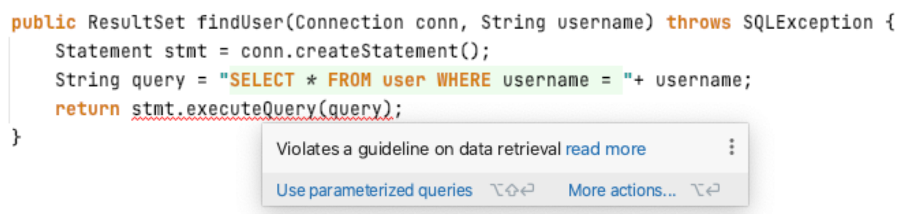
```
public class LandingPage extends AppCompatActivity {
```

Figure 7.10: The information error level marking is clearly visible but at the same time non-intrusive, as this is a permanent marking that can not be resolved.

Permanent markings, that remind developers of security implications of their decisions, should be marked as information. To continue on the example of private and public activities, in the code file that implements the activity, we mark the class definition at the information level. Hovering over the marking informs the developer whether the activity is configured as public or private, and provides a direct link to detailed information about the security implications. This marking is shown in Figure 7.10. Note how the markings are clearly visible and noticeable, but at the same time non-intrusive to developers already used to their IDE flagging code fragments.

The marking of code is accompanied by three descriptions. The information in these descriptions is important to ensure the continued use of the tool [103, 105]. Developers build trust with analysis tools, and this trust is quickly lost if they do not understand the tool's output [108]. The first description is the short error description, i.e., the text that appears when the developer hovers their mouse pointer over the marked code. It should be just one line. The purpose of this description is to attract attention, inform the developer that something should be addressed, not to explain how to address it.

We have learned through user feedback that it is most effective to attract the user's attention by starting with the "why" [109], the reason the code is marked and should be addressed. In the past, the short description used to indicate the possible vulnerability class, for example "Could lead to SQL injection". We believed that starting with the potential consequences, makes the developers realize the severity of their mistake and



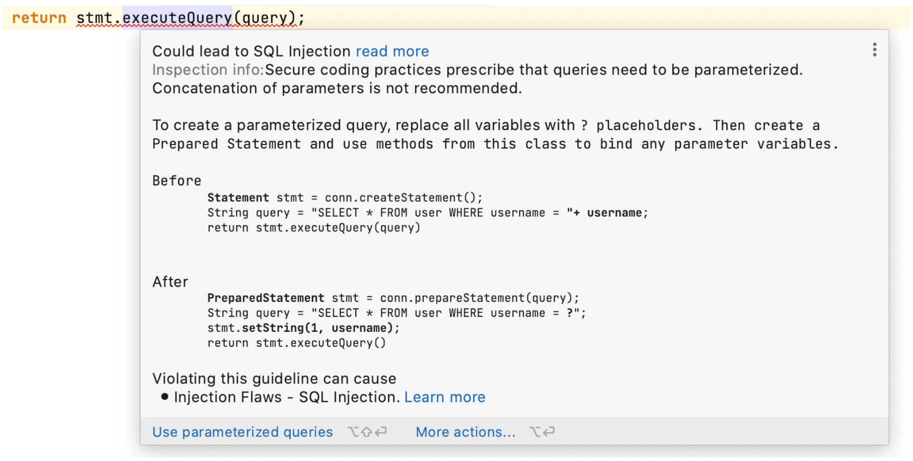
```
public ResultSet findUser(Connection conn, String username) throws SQLException {
    Statement stmt = conn.createStatement();
    String query = "SELECT * FROM user WHERE username = "+ username;
    return stmt.executeQuery(query);
}
```

Violates a guideline on data retrieval read more

Figure 7.11: The short description of a recipe is visible when hovering over a marking. It should attract the developers attention but avoid security jargon. Instead, the developer can be reminded of guidelines that are in place.

encourages them to immediately address it. However, as explained in Section 1.2.2, security jargon should be avoided when communicating to developers. The feedback in all of the descriptions should be targeted at developers, and hence the focus should be on the guidelines that were put in place, on the paved path. A better short description is hence, “Violates a guideline on data retrieval”, as depicted in Figure 7.11.

Next to the short description, a “read more” link is created by the IDE for the interested developer. Upon clicking this link, a pop-up is opened to show a more elaborate Hyper Text Markup Language (HTML) page. This is the second description. Figure 7.12 shows an example. This description is called the full coding guideline. The page starts with a short abstract, stating in one sentence what should be done, such as “Secure coding practices prescribe that queries need to be parameterized”. The page’s next section presents in detail what it means to use parameterized queries and gives an overview of the approved API methods. Small code examples are included as well, since previous research has shown examples are the fastest way for developers to understand a problem [105]. The goal of this description is after all to help developers find out quickly how to comply to the coding guideline without spending much time or effort. This is crucial for a security tool to feel well integrated into developer workflows. The last section of the description contains a list of possible consequences when the developer fails to address this issue. There is no mention of vulnerabilities or exploits until this point. Each item in the list contains a link to the SCW training platform to learn more about the vulnerabilities and how they are exploited. This way an interested developer (with too much time on their hands?) can still easily find the necessary information to learn the details of each vulnerability and the possible attacks. Following this training would require a context switch and would likely hurt developer productivity. In



```
return stmt.executeQuery(query);
```

Could lead to SQL Injection read more
Inspection info:Secure coding practices prescribe that queries need to be parameterized. Concatenation of parameters is not recommended.

To create a parameterized query, replace all variables with `? placeholders. Then create a Prepared Statement and use methods from this class to bind any parameter variables.`

Before

```
        Statement stmt = conn.createStatement();
        String query = "SELECT * FROM user WHERE username = "+ username;
        return stmt.executeQuery(query)
```

After

```
        PreparedStatement stmt = conn.prepareStatement(query);
        String query = "SELECT * FROM user WHERE username = ?";
        stmt.setString(1, username);
        return stmt.executeQuery()
```

Violating this guideline can cause
- Injection Flaws - SQL Injection. Learn more

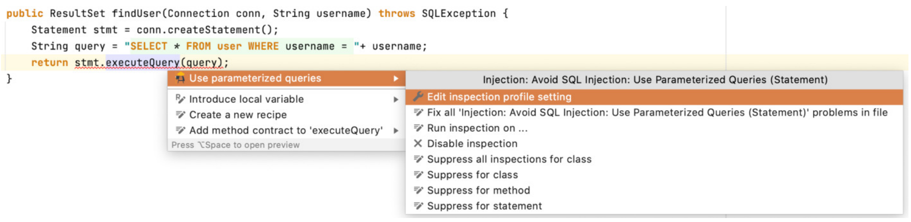
Figure 7.12: The full coding guideline is a more elaborate HTML page that explains the guideline in more detail and provides code examples.

the future the integration between the SCW training platform and the Sensei IDE plugin can be improved as described in the perspectives in Section 5.2.2.

The third description is visible to the developer when they press the IDE's key combination to start resolving the issue. A drop down menu appears, containing the possible quick-fixes' descriptions. IntelliJ also provides options to disable inspections locally or globally. Figure 7.13 shows an example. In this menu we provide a very short description of the actions that will be performed when this code transformation is chosen, such as "Use parameterized queries". A brief yet descriptive quick-fix description allows developers to decide quickly whether the fix is appropriate for them. If the effects of applying the quick-fix are well understood, the developer will trust the tool and apply the fixes more often. Sometimes the developer needs to choose between different possible solutions. However, it is advised to keep the number of fixes as low as possible, as to not complicate the issue unnecessarily.

7.2 Recipe features

Over time, several advanced features in the recipes have been developed following user feedback. In this section, I explain the problems they tackle and provide code examples for each one.



```
public ResultSet findUser(Connection conn, String username) throws SQLException {
    Statement stmt = conn.createStatement();
    String query = "SELECT * FROM user WHERE username = "+ username;
    return stmt.executeQuery(query);
}
```

Use parameterized queries
Introduce local variable
Create a new recipe
Add method contract to 'executeQuery'
Press ⌥Space to open preview
Injection: Avoid SQL Injection: Use Parameterized Queries (Statement)
Edit inspection profile setting
Fix all 'Injection: Avoid SQL Injection: Use Parameterized Queries (Statement)' problems in file
Run inspection on ...
Disable inspection
Suppress all inspections for class
Suppress for class
Suppress for method
Suppress for statement

Figure 7.13: The quick-fix description briefly describes the actions that will be performed when each option is chosen. IntelliJ also offers a feature to suppress markings of any inspection (Sensei or otherwise).

7.2.1 Lowering effective false positives

It is important to choose the right error level for the developer to pay attention to the markings, but also not to overwhelm them with markings to the point that they start to ignore them. Since the recipes can be created by anyone in the team, they should not be too obtrusive. To a recipe-writer, a false positive is an incorrect marking of code that is not violating a coding guideline. However, to a developer, a false positive is any code marking that they do not intend to fix and ignore instead [110]. A false positive from the perspective of the developer is also called an Effective False Positive (EFP) [107]. To ensure the usability of the tool, the EFP rate should be sufficiently low [107, 111].

To demonstrate EFPs we take a look at Operating System (OS) Command injection. One of the APIs that is banned in the OWASP ESAPI guidelines is `Runtime.exec`. This API is used to execute OS commands in Java programs. When user input is added to this command, OS command injection is possible and the attacker can gain access to the underlying OS. Using this information it is possible to create a recipe that marks all uses of the `Runtime.exec` method. While this is a good coding guideline, a security conscious developer recognizes that the method needs user input before it can lead to OS command injection. In rare occasions an OS command might be necessary for functionality and perfectly valid and secure. For example, launching another software product can be done securely as long as the command is hard-coded. An example of insecure usage of the `Runtime.exec` method can be found in Listing 7.1, examples of secure usage in Listings 7.2 and 7.3. The two secure examples are still violations of the above coding guideline, and they get flagged. An experienced developer has no intent to fix them, meaning they are two cases of EFPs. Notice how this EFP depends on the knowledge and skill of the developer, meaning that it might be beneficial to adjust the feedback of the tool for individual developers, as explained in the perspectives in Section 9.2.

In order to keep the EFP rate sufficiently low, we have introduced the concept of *trusted input*. Hard-coded input is automatically trusted, since a user can not influence it, and hence it can not lead to a vulnerability. However, function parameters and variables from other origins are considered untrusted by default. This is still in line with the philosophy to protect methods from future use, where we want to flag violations that can one day lead to vulnerabilities. The requirement in recipes of untrusted input can be added to arguments, this can be done using the YAML syntax or by using the GUI as shown in Figure 7.14. The next step is to define trusted sources. This also avoids the EFP in Listing 7.3. The resulting recipe can be seen in Listing 7.4.

```java
public void executeCommand(String command){
    Runtime r = Runtime.getRuntime();
    r.exec(command);
}
```

Listing 7.1: if the `command` variable contains unsanitized user input, this function is vulnerable to OS command injection.

```java
public void executeCommand(){
    Runtime r = Runtime.getRuntime();
    r.exec("explorer.exe");
}
```

Listing 7.2: Using a hard-coded command avoids the possibility that the variable will ever contain unsanitized user input.

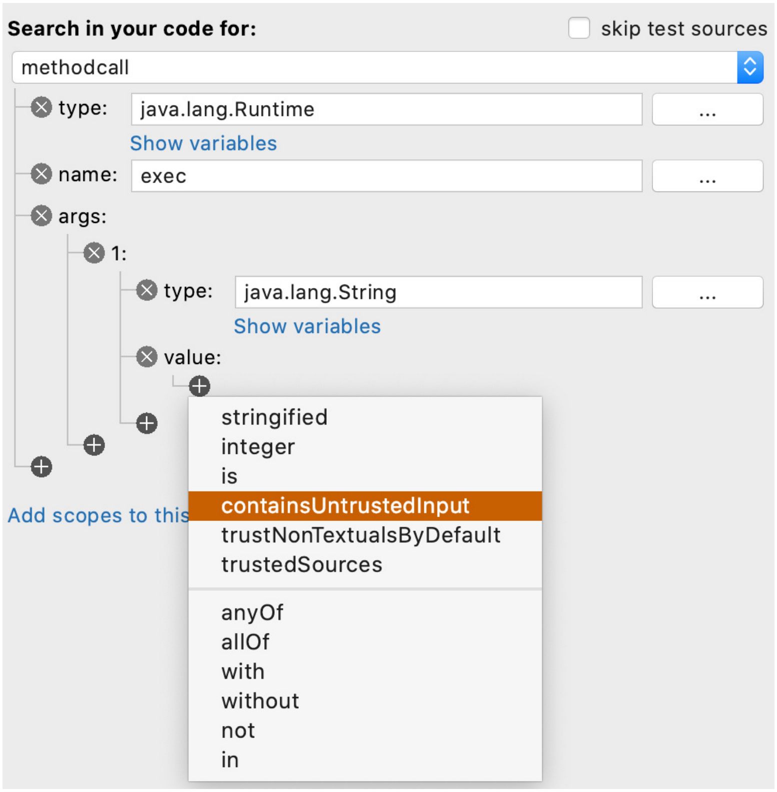
```java
public void executeCommand(){
    Runtime r = Runtime.getRuntime();
    String command = getSafeCommand();
    r.exec(command);
}
```

Listing 7.3: This code fragment is secure if the `getSafeCommand` method can be trusted to never return variables containing unsanitized user input.

```yaml
search:
  methodcall:
    name: "exec"
    type: "java.lang.Runtime"
    args:
      1:
        type: "java.lang.String"
        containsUntrustedInput: true
        trustedSources:
        - methodcall:
            name: "getSafeCommand"
```

Listing 7.4: This recipe trigger requires the argument of an `exec` methodcall to contain untrusted input before it will mark the methodcall. It also specifies the methodcall `getSafeCommand` as a trusted source of input.

Another way to allow recipe-writers to create more targeted recipes and to keep the EFP rate low, is to provide *trigger scopes*. Trigger scopes can be added by using the `in` keyword in the YAML syntax, or by using the GUI as shown in Figure 7.14. The in-clause can define restraints on the context. This makes it possible to create a recipe that prevents



type: java.lang.Runtime
name: exec
type: java.lang.String

Figure 7.14: It is possible to add requirements like untrusted input or an in-clause through the GUI.

the usage of `Runtime.exec` except in a class with name `AppLauncher`. Scopes like this can also help with performance, i.e., meeting the real-time checking requirement, since recipes that are out of scope can be skipped during analysis.

In the old editor, rather than *trigger* scopes, *recipe* scopes were a property of the entire recipe. They were added by selecting the type of scope and filling in some fields. By migrating these scopes to the YAML syntax, the scoping of recipes becomes more flexible. The recipe scopes that can be migrated to trigger scopes are the class scope, method scope, file scope, Android context scope, and Android build property scope. Descriptions of these scopes can be found in Appendix E.

There are two more recipe scopes, for which migration to trigger scopes is not useful. The *project scope* allows us to enable or disable recipes based on the name of the project. This is useful when different cookbooks are required for each project in a company. This scope is

```java
Cookie myCookie = new Cookie("secure", "success");
response.addCookie(myCookie);
```

Listing 7.5: This cookie is not configured before it is added to the response, as a result this code fragment is insecure.

```java
Cookie myCookie = new Cookie("secure", "success");
myCookie.setSecure(true);
myCookie.setHttpOnly(true);
myCookie.setDomain("sub.domain.scw.com");
myCookie.setPath("more/narrow/path");
response.addCookie(myCookie);
```

Listing 7.6: Several configuration options are added to narrow the scope that the cookie can be used, and to ensure it is not sent over plaintext.

no longer needed since we now allow cookbooks to be stored in the project, which is a lot more convenient then adding a scope to each recipe separately. The *library scope* can be used to enable recipes based on the presence of a library. This way we can disable a recipe if the fix uses a library that is not used in the project. Since this scope is created for the fix of the recipe and not the trigger, it cannot be added to the YAML syntax and remains a property of the entire recipe. In the future it could be useful to add scopes to quick-fixes, so that a recipe can provide different fixes depending on the presence of a library.

7.2.2 Support for libraries

Often quick-fixes are small code changes, such as adding a preceding method call or changing a parameter, but sometimes they involve more elaborate pieces of code. An example for this is adding a cookie to a Hyper Text Transfer Protocol (HTTP) request. Before adding the cookie, it needs to be properly configured. Insecure and secure code examples are shown in Listings 7.5 and 7.6, respectively.

If this fix is applied at multiple locations in a project, it can result in code bloat. It might then be better for the company to provide a method that replaces the original `addCookie` method. In this method the cookies can be first configured properly before calling the original `addCookie` method. Such a replacement wrapper method is shown in Listing 7.7. The new guideline for cookies is then to replace the `addCookie` method with the `safeAddCookie`, as shown in Listing 7.8. The creation of such a wrapper library is strongly recommended by the paved path method-

```java
public void safeAddCookie(Cookie myCookie, HttpServletResponse response){
    myCookie.setSecure(true);
    myCookie.setHttpOnly(true);
    myCookie.setDomain("sub.domain.scw.com");
    myCookie.setPath("more/narrow/path");
    response.addCookie(myCookie);
}
```

Listing 7.7: A wrapper library can be created to avoid code reuse and to improve clarity of the guidelines for the developer.

```java
Cookie myCookie = new Cookie("secure", "success");
safeAddCookie(myCookie, response);
```

Listing 7.8: Migrating to the wrapper library consists of replacing the original methodcall with one from the library.

ology, as the resulting guidelines are easy to understand for developers. At the same time any security bugs are confined to the *implementation* of the wrapper library, and no new bugs can be introduced by *using* the wrapper library. This makes the job of the security team easier as well.

The first example, where the cookie is configured properly, is a *library usage recipe*. This type of recipe provides guidance on using a library correctly. The trigger of the recipe is on APIs from the library. Code fragments are refactored without involving additional libraries, only libraries that are also used in the trigger.

The second example, where the insecure code is replaced with API calls from a different library, is a *library adoption recipe*. Instead of providing guidance on the correct usage of the APIs, such recipes promote the adoption of a new library. This type of recipe has a trigger in one library but their fix uses a different library.

As a proof of concept for library adoption recipes, support has been developed for the OWASP ESAPI library. Among others, the OWASP ESAPI contains replacement methods for commonly used insecure Java Development Kit (JDK) methods, the so-called OWASP ESAPI banned APIs[4]. To support the OWASP ESAPI in companies that adopt it, a recipe set was created to enforce the replacement of the banned APIs with their alternatives from the OWASP ESAPI. Feedback from these companies showed that this set of library adoption recipes was intuitive and easy to use for developers. Importantly, it increased the speed of

[4]https://www.owasp.org/index.php/ESAPI_Secure_Coding_Guideline

the library's adoption.

Library usage recipes are generally applicable to codebases because the trigger and fix of these recipes use APIs from the same library, thus ensuring that the recipes never mark any code when the fix is unavailable. The trigger and fix for library adoption recipes depend on different libraries. This implies that an applied quick-fix can result in the use of unavailable APIs.

Library scopes can be used for this purpose. When the library used in the quick-fix serves as a scope of a recipe, it will not mark any code if the library is unavailable.

7.2.3 Support for detecting design flaws

As discussed before, coding guidelines are enforced through mostly local analyses. This allows Sensei to intervene earlier in the development process and makes it possible to perform the analyses in real time while the developer is typing. For this reason the focus of the approach is mostly on implementation bugs. Detecting design flaws in the application code (rather than in the APIs it relies on) is typically harder. Still, we have learned that various flaws can be tackled through the use of configuration files and the previously described trigger and recipe scopes.

An example of a flaw that is difficult to detect with local analyses is excessive security logging. It is crucial to log important security events, but too much logging can make it difficult or impossible to locate certain events. While enforcing guidelines can help with logging securely (e.g., not logging sensitive data, not logging unsanitized input, logging in proper format) it is difficult to monitor the frequency of the logging through local analyses. Other examples are the use of transport layer security, or whether authorization is needed or not.

One scenario that, to the contrary, enables us to detect some design flaws, is when popular frameworks are used to implement security features. For example, enforcing the use of transport layer security in an Android app is as simple as adding a line to the Android manifest about clear text traffic. This can be seen in Listing 7.9, line 3.

Another example is the use of encryption. It is trivial to detect if a deprecated encryption algorithm is used by means of a known API, but it is harder to detect the absence of encryption through local analyses. One interesting class of mistakes that we observed was developers XOR'ing data, or encoding data, rather than encrypting it. As a solution, a coding guideline can be created that requires functions whose name contains "encrypt" to perform encryption through some of their approved API

```
<application
    android:label="@string/app_name"
    android:usesCleartextTraffic="false">
```

Listing 7.9: When the attribute `usesCleartextTraffic` is added to the Android manifest with value `false`, the Android OS will ensure that transport layer security is used for the communication with this application.

methods. If such a function *only* performs encoding or XOR operations, it implies that the required API calls are missing. In that case the tool suggests quick-fixes that insert the necessary API calls. These quick-fixes are only partial fixes: They inject code that invokes encryption routines on an unspecified string or byte array. It is then still up to the developer to remove the XOR operations and fill in the correct string or array identifier. This emphasizes again why it is beneficial to provide fixes in the IDE during development time. When this recipe is used during the development of a new method, the developer starts by creating a function declaration. When a function exists with "encrypt" in the name and an empty body, this is marked, as the required API calls are missing. The fix then inserts the required API calls, leading to comfortable and intuitive security help for the developer. This recipe is a good example to demonstrate the paved path methodology. It guides the developer along a predetermined path laid out for them to implement encryption securely.

We have also improved context-awareness to detect flaws by adding more recipe scopes. One such example is the Android *context scope*. As explained earlier, in the Android manifest a developer can configure capabilities of components such as activities and broadcast receivers regarding their communication towards the OS. They can listen to any other application, only to authorized applications, or only to the own application. The Android context scope allows us to enable recipes based on the configuration of the relevant component, so that we can enforce different recipes for different levels of exposure. Such a recipe can allow communication of sensitive information between classes that are configured as private components, but not between other classes.

7.2.4 Testing recipes

Testing custom recipes is a challenge. When new recipes are created, the recipe-writers first have to test the behaviour of the recipe manually.

They develop a few code fragments they expect to be marked, as well as some that should not be marked by the recipe. They then apply the transformations and manually inspect the resulting changes. The recipe wizard helps speed up these tasks by providing preview panels during recipe creation. A recipe-writer at a company, however, cannot be asked to perform the manual checks again every time they install updates to our plugin (including its underlying analyses) or to the IDE itself. Such updates always risk altering the outcome of a recipe. Instead automated testing is needed.

Such testing is available to the plugin developers at SCW. They have sufficient capabilities to automate unit tests that verify the behavior of the analyses. To do this, they also create a few demo recipes and test the behavior of these recipes. Since they have access to the code of the plugin, they can simply write unit tests and (directly) call internal plugin and exported IDE methods to test the markings and transformations and to compare them to the expected results. In other words, they can write code snippets that verify whether or not updates to tools alter the deployment of existing recipes. Such testing is unavailable to the custom recipe-writers in a company, however, which do not have access to, and definitely do not want to learn, the internal plugin APIs.

A better testing framework is hence required, such that the recipe-writers in companies can indicate the expected outcomes of every recipe they wrote on a number of code samples.. If we allow recipe-writers to define tests in the plugin itself, and store them in the cookbooks, these tests can automatically be performed when loading a new cookbook and after every IDE and plugin update. We can then notify the user when one of the recipes is no longer working as expected.

In the new recipe wizard it could be possible for the recipe-writer to select one or more of the examples that are marked in the preview panels and allow these examples to be used as the recipe tests. This way the recipe-writers are creating tests for their recipes with nearly no additional cognitive effort. However, this comes with the problem that (possibly confidential) code of the client is then stored in these recipe tests. At this point in time, implementing the necessary support for recipe-writer defined tests remains future work.

Chapter 8

Experiments and observations

Based on our own experience and anecdotal evidence, we have set goals and priorities during the development of Sensei. During my research, I conducted various experiments and interviews to asses these priorities, and to evaluate the various features as described in the previous chapter. In this chapter, I describe the goal of each experiment, its set-up, and report the findings.

If nothing else, take away from this chapter...

I conducted a controlled user experiment that showed Sensei markings are easy to understand and quick-fixes are applied frequently. Addressing security problems like this with Sensei, only caused an average increase in development time of about 11%.
Interviews with security professionals showed that the tool can be used effectively in a professional setting to detect and remediate security problems. They describe the customization of recipes as being easier and faster compared to other security tools. Nonetheless, usability experiments showed that the YAML code used for recipes can be overwhelming and that all users prefer the User Interface (UI) view of the new recipe editor. Creating new recipes was more successful if the users followed along with documentation, or if they looked at example recipes first, creating recipes from scratch can still be difficult.
The biggest disadvantages of Sensei compared to other tools are its lack of reported metrics and its poor integration in the Continuous Integration and Continuous Delivery (CICD) pipeline.

8.1 Controlled empirical usability experiment

In November 2018, I conducted an experiment with students to evaluate some features of the Sensei IDE plugin. This experiment has been well designed with the help of my supervisors and prof. Riccardo Scandariato who has practical experience with similar experiments. It was conducted with the help of the lecturer, dr. ir. Mattias De Wael, and two of my colleagues at SCW, Downey Robersscheuten and Tim Dekiere.

8.1.1 Goals and research questions

The main *goal* of the experiment is to observe the impact of the Sensei IDE plugin on developers and their code. The *purpose* is evaluating the usability and effectiveness of several of the features offered by Sensei during development, such as customized guidelines and quick-fixes in the IDE. The *quality focus* is the ability of the plugin to help developers adhere to secure coding guidelines without causing a significant cognitive burden. The study evaluates the behaviour of a developer. We aim at measuring the increase in cognitive burden when developers use the plugin, by measuring the impact on development time. In our experiment, we evaluated the impact on a group of students who have a consolidated minimum level of expertise in both web application development and web application security.

The above goal can be achieved by means of an experiment aimed at answering the following four questions:

- **Q1** How effective are the Sensei code markings at grabbing the developer's attention?
- **Q2** Does the plugin significantly impact the development time?
- **Q3** Do developers often use the provided remediation (quick-fixes) to resolve code markings?
- **Q4** Are there any specific code markings that significantly impact the usability compared to others?

8.1.2 Experimental set-up

Subjects

The subjects for this study are a group of third year students following the bachelor program for Computer and Cyber Crime Professional[1] at

[1] `https://www.howest.be/en/study-programmes?s_filter=bachelor`

the college Hogeschool West-Vlaanderen (Howest) in Bruges, Belgium. All students are in the third, and final year of the bachelor program. The experiment was performed in the context of the Secure Object Oriented Architectures class. In this course, the students are taught design patterns, how to design three-layered applications, and Java technicalities. During the entire course the focus is on development while conforming to Oracle's Secure Coding Guidelines[2].

All of the students are familiar with Java programming in IntelliJ IDEA, as it is the main language and IDE used in the education program. They are, however, not experienced or trained with the Sensei tool. This experiment was their first exposure to the tool.

The experiment was preceded by a secure coding tournament using the SCW platform. The goal of this tournament was to both engage the students as well as measure their skill level. Student participation was voluntary, out of the 75 students that participated in the tournament, 60 also participated in the experiment itself. However, only 32 students successfully submitted all necessary files after the experiment, as will be described further down this section.

Development task

The subjects were given a development task to complete with the Sensei plugin installed in their IDE. For the assignment they received the incomplete code for an employment web app. The application provides employees of a company a way to view, download, and upload their payslips, as well as to submit requests for absence. The application is written in Java and uses JSP as the server side technology. Some of the features are incomplete and must be completed by the subjects during the experiment. During the implementation of these features, the subjects are at risk of introducing a number of web application vulnerabilities. Below is a list of features to be completed and their associated risks.

- A web page to view absence requests: risk of XSS.
- A web form to search for absence requests in the database: risk of SQL injection.
- A web form to upload payslips in XML format: risk of XML injection, XML external entity, unrestricted file upload, and local file inclusion.
- Log all attempts made on the sign-in page: risk of log forgery.

[2]`https://docs.oracle.com/cd/E26502_01/html/E29016/scode-1.html`

Treatment

The treatment consisted of two parts. First, all subjects participated in a SCW tournament. The next week, all participants were given the Sensei IDE plugin, but some features were disabled for the control group.

Tournament One week preceding the experiment, all subjects participated in a tournament on the SCW platform. The subjects were handed 24 secure coding exercises in Java using JSP to complete within 90 minutes. The awarded points on completion of an exercise depends on its difficulty and the performance of the subject, as described in Appendix A. The scoring method chosen was "Forgiving".

The total maximum score for all exercises in the tournament was 5000 points. The highest score reached was 4920, while the lowest was 1600. The mean score reached by the participants (n = 75) was 3659 (s = 710). The mean time spent solving all exercises was 53.70 min (s = 16.07 min). Both the score and the time spent are approximately normal distributions as shown in Figure 8.1 and Figure 8.2. The score reached by each subject in this tournament was used to split the subjects into two equally skilled groups, a control group and a test group. The subjects were told that they were participating in an experiment regarding the Sensei plugin. They were told that they were split in a control group and test group but were not informed of which group they were part, or what the difference in treatment would be.

Sensei To complete the programming exercise, the subjects on both groups were allowed to use their own device and OS but they had to develop using the IntelliJ IDEA with the Sensei plugin installed. The Sensei installation of both the control group and the test group included a set of carefully tailored recipes to prevent introduction of the vulnerabilities described in Section 8.1.2. However, for the control group the markings and programming aid were disabled and the plugin was only used as a monitoring tool. All features to view, edit, or disable recipes were hidden, so that none of the subjects were able to consult or alter the recipes. The information available to the subjects, in the different descriptions, was designed as outlined in Section 7.1.4. In fact, the example given in Figure 7.12 is a guideline used during the experiment.

Ethical review board

The teaching staff proposed the experiment to the college's ethical review board. We helped them in writing a detailed explanation of the

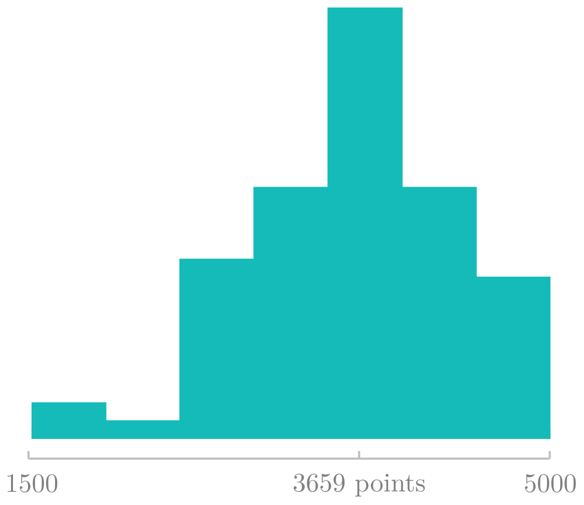


Figure 8.1: The points scored by the subjects during the tournament is approximately a normal distribution around the mean of 3659 points.

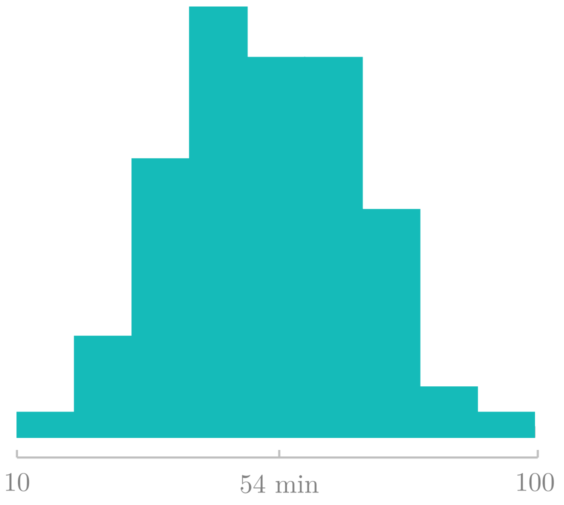


Figure 8.2: The time spent by the subjects (n=75) competing in the tournament is approximately a normal distribution around the mean of 54 min.

activities and the goals of the experiment. The board approved the experiment under two conditions. Firstly, experiment participation was to be voluntary and students were not to receive extra credit upon participation. Secondly, all data handed to the researchers was to be made completely anonymous. We and the teaching staff then operated in line with these conditions.

Experimental procedure

The experimental procedure is split into two main activities, the controlled experiment itself and the post-experimental information gathering.

Controlled experiment All subjects were allowed to use their own devices, and any resources they would normally use during development, such as books and internet access. We did not allow communication with other subjects. Subjects were allowed to take breaks and leave the room. However, as to not give incentive to finish hastily and without care, all subjects were required to be present during debriefing. The subjects installed Sensei by adding a custom repository instead of through the JetBrains Marketplace, as this allowed us to customize the features of the plugin for this experiment.

The subjects were given:

- a consent form to acknowledge that their data will be analysed anonymously;
- a repository Uniform Resource Locator (URL) to install the Sensei plugin, which automatically includes the set of recipes for each subject;
- a link to an archive containing the IDE project for the assignment;
- plugin installation instructions;
- a detailed description of the programming assignment.

During the controlled experiment, we asked the subjects to complete the assignment using the procedure below.

1. Open the plugins menu in the IDE and copy-paste the URL to the plugin repository.
2. Install the plugin and restart the IDE after the process has completed.
3. Verify correct installation of the plugin by finding the “Sensei” menu in the menu bar of the IDE.

4. Download the archive containing IDE project.
5. Extract the archive and open the project in the IDE.
6. Execute the project and read the messages in the console.
7. Open a web browser and browse to `localhost` to verify that the project is running correctly.
8. Sign in using provided credentials and get familiar with the functionality of the web application.
9. Read the description of the features to be implemented.
10. Complete the programming assignment in silence.

Throughout all phases of the experiment, we provided assistance to the subjects and answered all questions unrelated to the security of their code or the information displayed by the plugin. Indeed, despite testing on several operating systems and IDE versions there were some setup issues to solve.

Post-experiment information gathering When the task had been completed or the allocated time ended, the subjects were instructed to:

1. navigate to the Sensei installation folder and find the Sensei events file, which contains a log of all the actions monitored by the Sensei plugin
2. archive both the events file and the source files into one archive
3. submit the archive to the teaching staff

The events file contains timestamps and guidelines for all logged events. Examples of event files are shown in Table 8.1, and Table 8.2. The events in these tables include newly introduced guideline violations (ADD) that are later in time removed (DELETE). In Table 8.2 violations are removed using quick fixes (FIX). The removal of a guideline violation leads to compliant code. Sometimes it is possible to detect this compliant code with a different Sensei recipe that is marked as a compliant counterpart (C_ADD). For example, a parameterized query is the compliant counterpart of a SQL injection. A compliant counterpart is available for the recipe in Table 8.1. For other recipes, such as the one in Table 8.2, no compliant counterpart can be created. This is not possible, for example, for a recipe that forbids the use of OS commands, in order to prevent OS command injection. All code devoid of OS commands is technically compliant to this recipe, but we cannot create a recipe that detects conscious compliance to the recipe. The events also include the opening of a description (DESCRIPTION). The events file does not

ADD
DELETE
C_ADD
ADD
DESCRIPTION
DELETE
C_ADD

Table 8.1: The Sensei events file lists all events chronologically. In this example, a recipe was violated twice (ADD). Both times the violation was subsequently removed (DELETED) and replaced by compliant piece of code (C_ADD). Before correcting the second violation, the description was opened (DESCRIPTION).

include code or code locations, as our clients do not want to expose this information.

Several days after the experiment, all data was handed to us by the teaching staff after having obscured all personal data. At this point we discovered that the hand-in procedure was not correctly performed by all subjects, as the majority of the subjects had handed in either the code or the events file but few handed in both as requested. We asked all subjects to hand in again, stressing to include both the events file and all source files, but few subjects submitted a second time.

Without the source code in addition to the events file, I am unable to verify whether the code is still functional, as simply removing the relevant pieces of code would also effectively remove all guideline violations. On the other hand, without the events file we cannot verify which impact the Sensei plugin had on the security of resulting code.

This means I do not have sufficient data to compare results from both groups to evaluate the *effectiveness* of Sensei on improving the security of the final code, but that was never the main objective of the experiment. With the data from the 32 subjects who handed in their events file, I can still evaluate the *usability* of Sensei, albeit with a smaller data set than intended.

Analysis method

Since the logs in the events file do not include file locations, we sometimes have to make assumptions on which ADD and DELETE events should be paired. On occasion, there are multiple guideline violations with the same recipe ID present in the code at a certain time, as is the

ADD
ADD
FIX
DELETE
FIX
DELETE

Table 8.2: In this Sensei events file the recipe was first violated twice and then both instances were fixed using the quick fixes (FIX), no compliant counterpart exists for this recipe. From this information it is impossible to assert which of the recipe violations was removed first.

case in Table 8.2. In this case, we cannot know for certain which of the two violations is fixed first. During our experiment, this was the case for 8% guideline violations, with the two ADD events on average 37.75 s (s = 44.05 s) apart. For the measurements of the time between adding the violation and removing it, we assumed that the violations were removed in the same order as they were introduced. For all of the cases eventually either both violations were removed or neither of them were. The aforementioned assumption hence has no influence on the mean removal time and only influences the standard deviation of the removal time.

We observed three exceptionally long removal times and inspected the logs to determine the cause. Two of the outliers had events regarding other recipe IDs in between the ADD and DELETE events and so the subject did not spend this time actively solving the guideline violation. For further computations of removal time, these two outliers are left out. In between the ADD and DELETE events of the third case there were a number of DESCRIPTION events with the same recipe ID. In this case, we can safely assume that the subject did indeed spend 3.54 min actively resolving the issue.

8.1.3 Findings

Guideline violations

On average, the subjects in the test group introduced 17.64 guideline violations, as shown in the top right of Figure 8.3. The best performing subject (in this regard) introduced 2 guideline violations. For this subject, the events log showed enough C_ADD events to assume that the subject completed at least the majority of the programming exercise.

The worst performing subject added 37 guideline violations. In this case the events log showed a large number of ADD and DELETE events for the same recipe ID, making us believe that the subject was rewriting the code a number of times. This can result from attempting to implement the code functionally correctly or from attempting to resolve the guideline violation. The absence of DESCRIPTION events in the log is strong evidence for the former.In the control group the average number of violations introduced is 24.7, as shown in the bottom left of Figure 8.3. The amount of violations introduced in this group is between 2 and 54, however this distribution is not statistically significantly different from the test group ($p = 0.11$).

After completion of the assignment, the test group had 0.22 remaining guidelines on average, as shown in the top right in Figure 8.3. Out of these subjects, 79% ($n = 12$) finished the assignment free of violations. The average number of violations left at the end of the assignment in the control group was 8.8 and only 6% ($n = 1$) of the subjects finished the assignment without any remaining violations. This is shown in Figure 8.3 in the bottom right. This difference in remaining violations between the two groups is statistically significant ($p = 0.00015$).

Resolving guideline violations

Out of all the coding guideline violations in the test group, 98.4% have been removed eventually. Out of the removed violations, 73.3% have been removed with a quick-fix. For the remaining removals, it is not possible to know the intention of the subject, i.e., whether the violations were resolved manually as the subject spotted them as violations or whether the removal was part of rewriting (or removing) the code for another reason, such as simply meeting the functional requirements of the assignment. The four unresolved guideline violations each violated one different guideline, so there was no particular guideline causing the majority of usability problems. One was violating a SQL query guideline and the others were violations of several file upload guidelines by the same user.

Out of the violations that were resolved, 89.3% were resolved within one minute, and 99.5% were resolved within three minutes. Only one case, previously discussed in Section 8.1.2, took 3.54 min to resolve. This subject did eventually not use the quick-fix to resolve the issue.

On average the subjects of the test group took 19.10 s ($s = 25.22$ s) to resolve an issue. This large standard deviation is explained by a large difference in removal time for certain guidelines, as can be seen

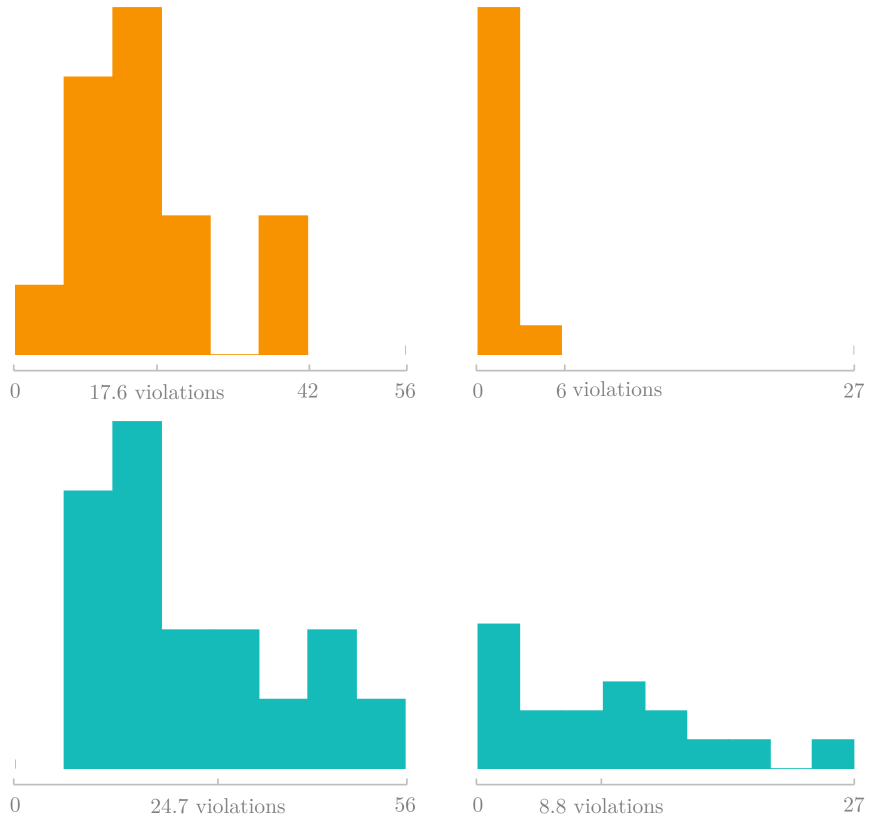


Figure 8.3: Histograms of the amount of violations introduced during the assignment (left column) and remaining at the end of the assignment (right column) by users in the test group (in orange) and the control group (in blue). The amount of violations introduced by the two groups is not statistically significantly different. The amount of violations left at the end of the assignment is significantly more for the control group.

in Figure 8.4. The mean remedation time when a quick-fix was used was 17.36 s. The violations that were removed by the subjects of the test group without a quick-fix were resolved in 21.73 s on average. The influence of the quick-fix on the remediation time is not statistically significant ($p = 0.7$). The average remediation time in the control group was 129.21 s ($s = 422.26$ s), and this is significantly different from the remedation time in the test group ($p = 0.001$).

On average more commonly known vulnerabilities such as SQL injection and XSS are resolved within less than 10 seconds, while the guidelines regarding file upload vulnerabilities take significantly longer. This is in line with general results from the trained 2PL model in the experiment of Section 4.1. The model showed that exercises about commonly known vulnerabilities such as SQL injection and XSS have a lower mean difficulty on the SCW training platform. Both these results indicate that understanding and fixing these common vulnerabilities is relatively easy. But in this experiment, as well as in practice, many developers still make those mistakes. This gap between knowledge and practice shows that when developers are focused on the functionality of their code, they can easily lose track of the security.

Besides familiarity with the vulnerability, the difference in speed for resolving the vulnerabilities can also be explained by the fact that one piece of code can violate multiple guidelines. This was often the case for the file upload guidelines, the naive implementation without any security checks violates guidelines regarding file path, file size, and file extension. The developers violating these guidelines receive a lot of simultaneous feedback, which takes longer to process. Fixing these vulnerabilities then also involves slightly larger pieces of code, as opposed to the often single line of code that needs to be fixed for the other guidelines. This is also in line with observations of the 2PL model, where the locality of the fix has a big influence on the mean difficulty of exercises.

All of the subjects used at least one quick-fix, with an average of 12.71 ($s = 4.73$) quick-fixes used per subject. Less than half of the users (42.85%) have opened a description. On average the subjects opened 2.79 ($s = 7.64$) descriptions.

Development time

Using the events file from Sensei we can determine the approximate development time for the entire experiment. If we take the time difference between the first and the last event, this will likely be close to the total development time. This can be done for all users of both the control and

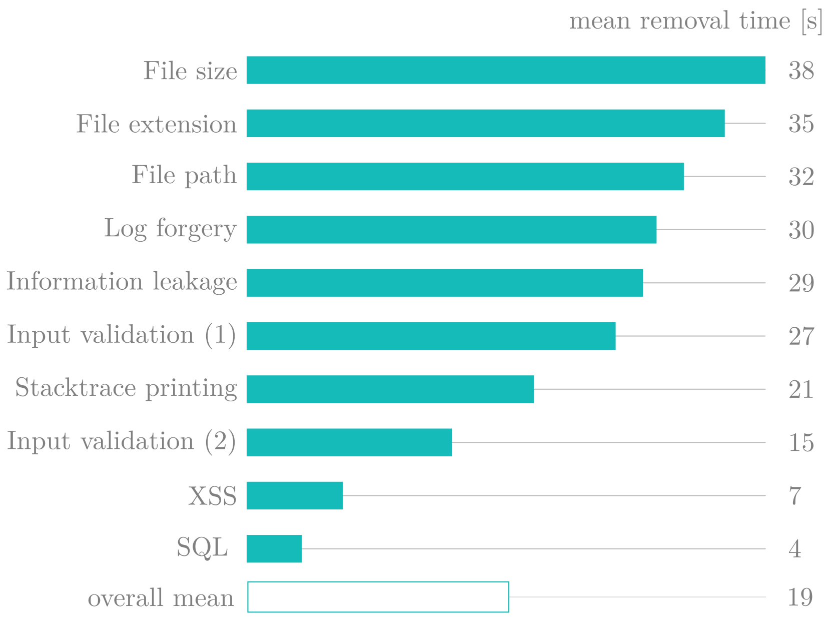


Figure 8.4: The average removal time for each guideline fluctuates heavily and many are different from the overall mean removal time of 19.10 seconds.

the test group that handed in the events file. For users in the test group we can also approximate the time spent addressing Sensei markings by taking the sum of all removal times. We can compare these results to see how much impact Sensei had on the development time. Users of the control group (n = 17) spent on average 61.54 min (s = 17.68 min) to complete the experiment. Users of the test group (n = 15) spent on average 68.75 min (s = 14.42 min). This is an increase of 11.72% in development time when using the plugin. The time spent by the test group addressing coding guideline violations was on average 8.42 min (s = 10.06 min). The average share of total development time that is spent addressing guideline violations is 11.28% (s = 12.25%). This is consistent with the previously measured 11.72% increase in development time. This increase in development time is explained by the large number of coding guideline violations that are not addressed by users in the control group. On average 8.8 violations were remaining in the control group, and the mean remediation time in this group was 129 s. Resolving all guideline violations would on then lead to an average development time of 80.46 minutes, which is 17% longer than the test group. This is a conservative estimate, as vulnerabilities that are detected and resolved in later stages of development are typically more costly and time-consuming to fix, compared to remediating guideline violations during development [112, 113]. In the perspectives in Section 9.2.3, I propose some designs for future experiments that could explore these results more thoroughly.

8.1.4 Threats to validity

In this section, I check the experiment against the possible threats to validity as proposed by Wohlin et al. [114].

Conclusion validity

The final score of each subject in the tournament is not a complete estimate of the subject's skills regarding security or secure development. Since there is a time limit, a good score is also partly achieved by time management. On one hand, taking too much time to complete the exercises will result in missed scoring opportunities by not finishing all exercises. On the other hand, answering too hastily may result in mistakes that otherwise could have been avoided, again resulting in a loss of points. However, the exercises were in the same language and framework as the development task, and the subjects also had a limited time to complete this task, so it is a reasonable estimate.

Each group of subjects were given the exact same development exercise, only different treatment.

The subjects were not heterogeneous, as they were all bachelor students, and the tournament score was used to avoid random irrelevance to some degree.

Internal validity

Before starting the experiment, we clearly explained the programming assignment and answered any arising questions publicly. The experiment itself was conducted in a single session, with all participants in the same room, this excludes all threats related to location, and repetitions.

Since the experiment was preceded by a secure coding tournament, and the experiment took place in a security oriented class, this history can affect the experimental results. However, do note that the entire bachelor's program followed by the subjects is focused on security, so the security related activities are not that different from usual day-to-day activities.

Since the experiment took over an hour, depending on the speed of development, subjects may react differently as time passes. Indeed, to avoid students getting tired, bored, or frustrated, we allowed them to take breaks and leave the room. We also note that the opposite is possible, and even likely, the subjects could have been learning and adjusting their behaviour during the experiment. This will also interact with the selection, since the test group receives feedback on their behaviour through the tool, and the control group does not.

The effect of letting volunteers take part in an experiment may influence the result, since they are generally more motivated and suited for a new task than the whole population. The subjects group might not be representative of the whole population.

Since some of the subjects did not hand in their Sensei events file, it can be useful to characterize the dropouts in order to check if they are representative of the total sample. However, due to the anonymity of the data, we were unable to do this.

The subjects in the control group are receiving less desirable treatments. As the natural underdog, they might be motivated to reduce or reverse the expected outcome of the experiment. This threatens the comparison in development time between both groups. This effect is expected to be more present if we had been comparing the security of the resulting code, but we did not do this. Moreover, we took the necessary precautions to avoid that the control group was aware of being in a less desirable situation, such as leaving them unaware of what the Sensei tool looks like, thus leaving them unaware of it being disabled for them.

Construct validity

We collected information about the time spent resolving issues as the time between introducing the violation and removing it. However, during this time window the subjects might still be working on the functionality of the code instead of its security. Since these two tasks are mostly interleaved, it would be nearly impossible to precisely asses the two times separately. Hence our focus on the increase in total development time as an additional measurement.

The subjects were aware that they were participating in an experiment. This in itself may make the subjects more receptive to its feedback.

The subjects were allowed to use any resource they desired to complete the task. This factor may influence the results, because better resources could help in completing the programming task faster or with less security issues.

The subjects might try to figure out what the purpose and intended result of the experiment is. They are likely to change their behaviour based on their guesses about the hypotheses. For this reason we did not disclose to the participants whether or not they were part of the control group or the test group. But it is likely at least the control group would realise their role in the experiment after not receiving feedback from the tool for a while. This does not influence our results about the interaction with the tool since the control group does not interact with it. The test group is less likely to realise their role in the experiment, but the realization is more likely to cause an effect.

Some people are afraid of being evaluated. A form of human tendency is to try to look better when being evaluated, this could influence how the test group interacts with the tool. It is possible that the subjects would ignore markings more often if they were not being evaluated.

External validity

All students come from the same college and the same bachelor's program. It is possible that subjects from a different college or program might result in different performance while completing the development task.

The subjects were not trained or experienced in the use of the treatment. It is possible that developers with more experience with security tools in general, or specifically Sensei, behave differently when interacting with the tool.

Since all subjects were tasked to develop a web application using Java JSP, the findings might not relate to development in general. The findings might not apply to development of other types of software, or when using other languages, or frameworks.

The subjects mostly lack professional experience, most of them only having done internships. It is possible that developers with more professional development experience behave differently.

8.2 User testing with individual developers

During my research of the Sensei IDE plugin, Sensei's product manager at SCW, Charlie Eriksen, has organized two usability tests. The first usability test was performed in October 2020, the second several months later in April 2021. The usability tests were executed by the company Haxor[3]. Afterwards, screen recordings and insights were shared with us to evaluate the tests.

8.2.1 Goals and research question

The main *goal* of the tests is to observe developers creating new recipes for the Sensei IDE plugin. The *purpose* is evaluating different features of the Sensei recipe editor. The *quality focus* is the ability of the plugin to allow developers to easily create the recipes they have in mind. The study evaluates the UI and User Experience (UX) of the recipe editor.

I aim to observe which features in the recipe editor are most effective and usable. In the experiment, I evaluated the behaviour of several groups of developers who have a minimum level of expertise in software development in Java.

The above goal can be achieved by means of an experiment aimed at answering the following four questions:

- **Q1** Which features are most useful when creating a recipe?
- **Q2** What are the main shortcomings when creating a recipe?
- **Q3** Which features are most useful when creating a quick-fix?
- **Q4** What are the main shortcomings when creating a quick-fix?

[3] `https://haxor.sh/`

The same usability tests are also used by Sensei's product manager to evaluate the installation process, the onboarding process, and the documentation. Those results will be briefly discussed as well.

8.2.2 Experimental set-up

Subjects

For both runs of the experiment, the goal was to have at least five subjects, a frequently used number in usability testing. It is the number of users needed to detect 85% of the problems in an interface, given that the probability of each problem occurring is 31% [115]. In practice, UI and UX problems do not affect users in a predictable way, and the probability that a user encounters a problem can be significantly lower. In that case a larger number of subjects is needed. However, it is advised to use an iterative design and test strategy, where five subjects are brought in to find problems and these problems are fixed before bringing in five more [115]. In order to guarantee successful tests for five subjects even in the case of a technical problem, each round six subjects were asked to participate, resulting in a total of 12 subjects.

All subjects are hired by Haxor from the United States and speak English. They are recruited from the Haxor Developer Community, DevPort, and online freelancing websites. The subjects are of entry and intermediate skill level and have a minimum of 2 years professional experience. All of the subjects have programmed in Java before and are familiar with the IntelliJ IDEA. An effort has been made by Haxor to choose subjects with varying backgrounds, and at least one subject with more than 5 years of professional experience is included in each test.

Task

The task was prepared by a UX design expert at SCW in cooperation with the product manager of Sensei and myself.

The developers were given a project with a few fragments of example code. These code fragments contain various calls to desirable and undesirable methods.

The subjects were tasked to create Sensei recipes and quick-fixes that transform the undesirable method calls into their desirable counterparts.

The required Sensei recipes are of increasing difficulty:

- Recipe 1 is already developed, users are questioned about their understanding of the recipe.

- Recipe 2 replaces the undesired methodcall by a different methodcall with the same signature (arguments and return type).
- Recipe 3 replaces the undesired methodcall by a different methodcall with a different signature (different arguments, same return type).

Experimental procedure

Testing procedure All subjects were allowed to use their own devices, and any resources they would normally use during development, such as books and internet access. To record their session, subjects were instructed to use Paircast[4]. Paircast is desktop software that records a developer's screen, microphone, code changes, and open applications as they work.

Developers were instructed to speak their thoughts out loud. In the first round of user testing, the subjects were also prompted questions after completing each of the assigned tasks. This turned out to be unnecessary as the subjects gave plenty of feedback without being prompted, hence the questions were dropped in the second round.

I reviewed all of the video recordings. I manually timed each action, as well as recorded any notable actions or comments made by the subjects.

8.2.3 Findings

Installation and use

All of the users who installed the plugin through the Plugins menu and the JetBrains marketplace have done so without any problems and within several minutes.

All users found it easy to understand existing recipes and apply the quick-fixes. Users claimed the recipes and quick-fixes looked exactly like the IntelliJ quick-fixes and that they would use them frequently.

Creating recipes and quick-fixes

When tasked to create new recipes, not all users had as much success, and the opinions were somewhat divided.

The instructions of the first usability test explained that Sensei recipes are stored in a local file called *rules.sensei*. When reading those instructions, several users opened this file to take a look. When, in the

[4]`https://paircast.io/`

next steps the users were then prompted to create new recipes, one of them did not look for the recipe editor, but instead began to edit this file at first. The other users who did find the recipe editor, opened it while the *rules.sensei* file was opened in the text editor. As a result, the preview panels did not show any relevant code examples. In the second usability test, the *rules.sensei* file was not mentioned and all users found the recipe editor immediately, and had relevant code files open in the preview panels instead.

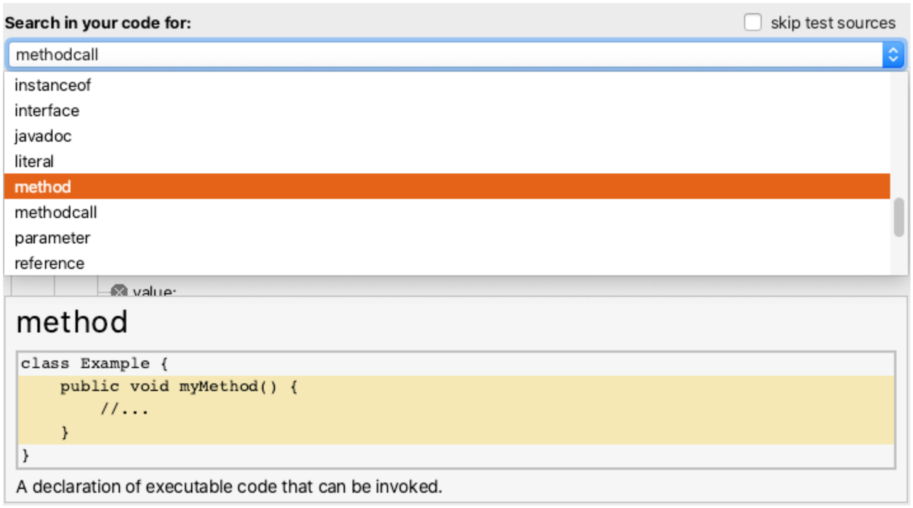
In the first usability test, only one user was able to find Sensei documentation by searching for it on the internet, 3 of the other users asked for more documentation when they could not find any. For the second test, the documentation was easier to find, and all of the users looked for it and found it. Some users in both tests took their time to read the documentation, they followed along with the getting started guide, for example. All of these users reported they found it easy to create new recipes. The users who opened existing recipes in the recipe editor before trying to make their own, generally had less difficulties creating new recipes than users who did not look at examples.

Many users used the context menu to open the recipe editor. However, in the context menu all of the users selected the option "start from scratch". For some users this was because their caret was not in a relevant position, others simply did not use the context-aware options when given the opportunity. It is possible that the different options in the context menu need to be more descriptive. It is also possible that better training in the form of documentation or an improved quick-start guide can resolve this usability problem.

When creating a new recipe in the recipe editor, some users were overwhelmed by the code view, all of them preferred to use the UI view. One user refused to create a new recipe, saying they do not want to learn a new language to do so.

Users who did not look at the documentation closely were more likely to be overwhelmed with the amount of options in the drop-down menus of the recipe editor. The options to choose from are not clearly enough described, not all users are familiar with the different syntactic components and their differences, e.g., method vs. methodcall. None of the users noticed the hints that describe each of the elements in the drop-down menu when they hover over it, as shown in Figure 8.5.

In the fix menu it is possible to reuse arguments of the original code through a template language. All users who made use of these templates, did so by copying the template from a different recipe and adjusting it to their needs. None of the users used the suggestions available in the



method

```
class Example {
    public void myMethod() {
        //...
    }
}
```

A declaration of executable code that can be invoked.

Figure 8.5: There are hints available that describe the different syntactic components that can be used in Sensei recipes. These hints are visible when hovering over the different options in the drop-down menu of the recipe editor.

fix menu.

8.2.4 Threats to validity

The focus of these tests is not to generalize any of the behaviours of the subjects, but rather to identify common usability problems in the interface of the recipe editor. Several of these problems were detected. I make no further attempts at interpreting the results from this usability test, I only report the findings as they appeared in the screen recordings. There would be many threats to the validity of any further conclusions drawn from the findings of these usability tests. The number of subjects is small and the tasks they were asked to complete are artificial.

8.3 Industry trial in 2018

One of the earliest customers of Sensei closely monitored their use of the tool during a trial period of several months in 2018. They reported their findings to us and at the end of the trial they purchased additional licenses for the tool.

8.3.1 Goal

The *goal* of the trial is for the client to observe the effects of the Sensei IDE plugin on its development process. The *purpose* is to help the client decide whether the Sensei IDE plugin is worth purchasing. The *quality focus* is on the time and money saved by detecting possible vulnerabilities early. The client aims at better estimating the Return on Investment (ROI) of the potential purchase. During the trial, they can both collect some objective data on the number of vulnerabilities prevented, as well as collect opinions from the application security team and the developers involved in the trial.

8.3.2 Set-up

Subjects

The client is a large bank included among the top 25 banks of the world as listed on wikipedia[5]. The subjects were a group of five full-time developers selected by the client for their security knowledge. The tool was also given to an employee responsible for application security to help evaluate the trial. This employee was our main contact during the trial period.

Tasks

The subjects are part of teams developing and maintaining the web and mobile applications of the client. They were developing in either IntelliJ IDEA or Android Studio. During the trial period they continued their daily responsibilities as usual, reporting periodically to the application security expert on their impressions of the tool.

Treatment

The developers were given two sets of recipes, one for general Java applications, and one for mobile Android applications in particular. The cookbook for general Java applications was developed by SCW developers in cooperation with the application security expert of the client. They advised what they wanted to achieve from the developers with the tool, and we created recipes to enforce this. The second cookbook was also developed by us and was based on the official Android developer

[5] `https://en.wikipedia.org/wiki/List_of_largest_banks`

guidelines[6]. All of the recipes in this set had scopes so they would only be active when the developer was working on an Android project.

Information gathering

The client did not share their code nor their Sensei events file. Our contact was given the ability to view the summary of the Sensei events file in the form of an update to the Sensei plugin that enables them to view the statistics on each device. Our contact at the company evaluated these and shared some of their insights as well as opinions from the subjects themselves.

8.3.3 Findings

They reported that during the trial, over 200 markings were found that were legitimate markings that could lead to vulnerabilities. With the majority of these present in legacy code, they were security defects already in production. The two most common categories were mentioned as being tapjacking and sensitive information leakage (mostly caused by leaking stack traces).

The subjects reported the tool as useful and not too intrusive when working on new code. They also reported improving their security knowledge, driven by the markings from the plugin.

After the trial, the client chose to extend their current licenses and purchase additional ones.

8.3.4 Threats to validity

There are many threats to the validity of conclusions drawn from the findings of this trial. We have no detailed knowledge or control over the task, the subjects, the time, or indeed over any other aspect of the trial. We are unable to account for any noise in the metrics or any conditions that limit our ability to generalize the results. For this reason we make no attempts at interpreting the results from this trial, we only report the findings as they were reported to us.

8.4 Industry interview in 2021

In August 2021, I had the opportunity to interview the security team at a large company that has been using Sensei for over two years. They

[6] https://developer.android.com/

shared their insights in how the tool is used, what they liked about it and what its biggest shortcomings are in their eyes.

8.4.1 Goal

The *goal* of this interview is to learn how Sensei is actually used in an industry setting. The *purpose* is to understand if the design goals as explained in this book align with the expectations of the users, and to observe which features are most useful and which are lacking or missing. The *quality focus* is on the frequency at which features are being used, and for which purpose.

8.4.2 Set-up

Subjects

The client is an international cloud computing company building and maintaining enterprise software used by more than 26,000 customers. The teams observed and interviewed are based in Europe. They are 8 teams of developers and a team of 12 security professionals. Most, but not all, security professionals have prior development experience, some at this same company.

Task

The subjects are part of teams developing and maintaining the software of the client. All of the Sensei users use the IntelliJ IDEA. During their use of Sensei, they have continued their daily responsibilities as usual.

Treatment

Use of the Sensei plugin in these teams is voluntary. About 90 employees in total are using the tool, of which 60 are using it more actively. Five of the security professionals use the tool, as they are the ones involved in Java development. The remaining users are developers.

The teams have been using Sensei for over two years. When it was purchased, one of the security professionals gave a presentation and a demonstration of the tool to interested coworkers. Most attendees were team leads, managers, and some security champions. Security champions are developers who show more interest, and higher competence regarding security. From there on, use of Sensei has not been actively promoted across the company. However, one security professional regularly discusses the tool in the security champions group meetings, as well as in the dedicated support channel on Slack.

The security team also uses Fortify, Checkmarx, SonarQube, Semgrep, and FindBugs. They have sufficient context to compare Sensei to other security tools. The listed tools, together with other comparable tools, are discussed in more detail in Chapter 10.

Information gathering

The team of security professionals is our contact at the company. Two members of the team agreed to a meeting in which I interviewed them on their use and their impressions of Sensei as well as other security tools they are familiar with. This interview was recorded and the recording reviewed before writing this report.

One of the interviewees has been with the company the entire time that Sensei was purchased, this person has prior development experience. The other interviewee joined the company after the purchase of Sensei, this person does not have prior development experience, but has been a security professional for a longer time.

8.4.3 Findings

Adoption

The security professionals found that adoption of the tool is not easy. It is hard to get a chance to show value to the developers, and they are hesitant to install new tools in their IDE. It is easier to convince the security champions who are more interested, and actively looking for tools that can help them produce secure code faster. So far, the security professionals have preferred the hands-off approach and allowed the tool to organically spread, instead of making it mandatory. The security professionals, many who have development backgrounds, are convinced that the tool can be an asset to developers outside of the context of security as well.

Recipes

The security professionals have created around 50 recipes. These are stored on a remote server and distributed to the developers as a read-only cookbook. The recipes in this cookbook are not all related to security, but around 70% of them are. The other recipes are related to quality and code conventions. Many of the security team have a development background, they have added these recipes in an attempt to show their value to the developers. No public cookbooks are used, but the recipes in the public cookbooks served as inspiration for their own custom recipes.

The company has a lot of clear coding standards that are published and used outside of their company as well. These coding standards are used as a basis for recipes and their quick-fixes. No real consultation with developers is needed, as it is generally agreed by developers and security experts that these coding standards are to be used. However, to create the quick-fixes, the security professionals frequently consult internally with more experienced team members. Since some of them are former developers at the same company, they have an intimate knowledge of the codebase.

The company uses many wrapper libraries. However, these are often not specifically written for security purposes only. Several Sensei recipes exist to migrate to wrapper libraries or to different versions of APIs.

The security team is currently unaware of the number of recipes that are being created by developers themselves. They are also unaware if developers are frequently remediating markings from the distributed cookbook. In fact, in their eyes, visibility into metrics like this is one of the biggest shortcomings of the tool.

Recipe editor

The security professionals create both recipes from scratch and from context and believe both use cases are important and necessary. They most frequently use the UI view to edit recipes, but to refactor recipes and make bulk changes, they sometimes use a text editor as well.

They believe the preview panels and the recipe editor are by far the most useful features of the entire plugin. These features make customization of the recipes significantly easier compared to the other tools they are using in the SDLC.

Descriptions are often used, but the full coding guidelines are not usually customized to the specific recipe. The coding guideline provided is instead a generally applicable description that provides links to documentation about the secure coding standards that are used at the company.

The security professionals use recipe scopes frequently. The scopes are used to limit recipes to certain packages and modules. This is only done as a consideration for developer usability, to avoid false positives in the large codebase.

Finally, they try to provide quick-fixes as often as possible, but admit it is not always possible. Sometimes, the quick-fix provided requires the developer to make additional changes.

Paved path methodology

The security team does not use the paved path methodology. However, their practices are in line with many of the goals of this methodology.

The security professionals try to be *enablers*, and not only tell developers what they do wrong, but also provide guidance as much as possible. They provide a service to developers and are aware of developer usability. The team prefers to neglect some parts of the security of the software over generating too many false positives, which could result in EFPs.

They believe Sensei supports this enablement approach through its quick-fixes. Since a complete Sensei recipe includes a quick-fix, the security team is forced to offer remediation guidance. This remediation guidance in turn enables the developers to resolve security markings by themselves.

For the security professionals without former development experience, this requirement of a quick-fix forces them to be closer to the development workflows. Sometimes, creating a quick-fix pushes the limits of their knowledge of programming. In that case, they do not consult the development team, as suggested by the paved path methodology, but instead consult with former developers in the security team itself.

Disadvantages

As mentioned before, visibility into the developers' practices with the tool is one of the biggest shortcomings of Sensei in the eyes of the security team. So far, features that report back information from the IDE have been avoided as we expected customers would be hesitant of such features. Many of the metrics that the security team requests are available in the IDEs of the developers, in the Sensei events databases. Clearly, gathering those databases is not a convenient way to collect that information. On top of that, currently, we provide no convenient way to visualize the results in a management dashboard, which other tools commonly do.

Alternatively, some metrics can be collected through server side scans. It is possible to run IntelliJ IDEA inspections from the command line, including the Sensei recipes. The resulting scans are not as efficient as those by standalone tools, such as static analysis tools discussed in Chapter 10. In particular, the security professionals reported that the memory usage is exceedingly big for large enough codebases. Since these interviews, the developers at SCW have fixed a memory leak in the command line scans which has mitigated this problem. It is also more difficulty to automate running IntelliJ IDEA inspections in the CICD pipeline compared to tools who provide better integrations for this purpose. The convenience of running scans in other stages of the SDLC seems to be the main reason the security team uses some of the other tools.

8.4.4 Threats to validity

There are many threats to the validity of conclusions drawn from the findings of this interview. We have no detailed knowledge or control over the task, the subjects, the time, or indeed over any other aspect of the use of Sensei. We are unable to account for any noise in the metrics or any conditions that limit our ability to generalize the results. For this reason we make no attempts at interpreting the results from this interview, we only report the findings as they were reported to us.

Chapter 9

Discussion and perspectives

In the previous chapter, I described the goal and the set-up of each experiment, and reported their findings. These findings allow us to evaluate different features of the Sensei plugin, as well as its use in the paved path methodology. In this chapter, I summarise the findings, explain the lessons we learned and how they can affect the development of Sensei in the future.

If nothing else, take away from this chapter...

Our findings and those of other research, show that customization of recipes can have a significant impact on the EFP rate, and hence the usability for the developer. Former research has shown that applying secure coding guidelines early in development has a positive impact on the codebase. As shown in the experiments of this work, customized recipes can improve adherence to such guidelines as quick-fixes are frequently used. If they are designed properly, applying the recipes regardless of context has minimal impact on performance, and helps improve code quality in many cases.

Security professionals report that creating these customized recipes with the recipe editor is easier than customizing rules of comparable tools. Despite this, usability tests revealed that some features of the recipe editor can still be improved such as the templating language to reuse parts of the original code in a quick-fix.

9.1 Discussion

9.1.1 Installation and first use

Because Sensei is distributed as an IDE plugin, it can be easily installed from the IDE itself, using features that many developers are familiar with. None of the developers in any of the experiments needed more than several minutes to install the plugin.

After installation, the startup time of the IDE is not measurably affected by the tool. Sensei only performs a license check, of which the duration is shorter than the measured variation in IDE start-up time.

In the early industry trial and the controlled experiment, customized Sensei recipes were provided by us as a service. In the usability tests and the most recent industry trial, the subjects were provided with public cookbooks that could be used as a starting point, but they were encouraged to create their own recipes.

In both groups we have observed developers who unknowingly addressed Sensei markings, thinking they were regular IDE markings. This is strong evidence that the tool is intuitive and feels like a natural extension of the IDE.

9.1.2 Recipes

Sensei is a developer tool first, and this is evident from the recipes that are used in industry settings. A significant portion of the recipes are related to the quality of the code rather than its security.

We have noticed, in practice, that security and quality are often closely related. High quality code is easier to understand and maintain, and hence also to secure. But often, writing high quality code can also lead to secure code in a more direct way.

Take the example of a SQL query. Writing a data retrieval method of high quality means that the query is easy to understand, but also that the data is retrieved at high speed. When the query is parameterized, the database can pre-compile a query plan, which speeds up the execution of the query. Of course, using parameterized queries at the same time ensures that the query is safe from SQL injection. Even if the current query did not use any (unsanitized) user input, using a parameterized query will protect it from future use. It will also set a good example for future developers writing similar methods. Developers will often copy existing code and make some changes to fit their needs. After all, developers try to be as efficient as they can in delivering code. If no parameterized queries are used, a subtle change can mean the difference

```java
public ResultSet getUserById(int id){
    String query = "SELECT * FROM user WHERE id = " + id;
    PreparedStatement stmt = this.conn.prepareStatement(query);
    ResultSet rs = stmt.executeQuery();
    return rs;
}
```

Listing 9.1: This method concatenates an integer value to the query. An integer variable can not alter the query, and hence this method can not lead to SQL injection.

```java
public ResultSet getUserById(String id){
    String query = "SELECT * FROM user WHERE id = " + id;
    PreparedStatement stmt = this.conn.prepareStatement(query);
    ResultSet rs = stmt.executeQuery();
    return rs;
}
```

Listing 9.2: This method concatenates a String variable to the query. As a result it is vulnerable to SQL injection.

between secure code or a vulnerability, as shown in Listings 9.1 and 9.2.

In industry, recipes were created, both to help developers use libraries correctly, as well as to migrate to new libraries. Security professionals reported that many wrapper libraries are used in their codebase to make the life of developers easier. These wrapper libraries were rarely developed solely for security reasons, but often did include security automation. It is clear that for such wrapper libraries, it is crucial that the recipes can easily be customized.

9.1.3 Recipe editor

When testing YAML syntax and creating recipes from context, we observed a significant speed-up in writing recipes. Even for users that were experienced with the old rule models, the preview panels in the new recipe editor and the context-aware suggestions greatly improve the recipe-writing process.

Security professionals report that Sensei is the easiest tool they have used when it comes to customizing recipes. The majority of comparable tools allow writing custom rules or analyses in one way or another, as described in the related work, in Section 10.4.5. Writing rules for these tools is often done through complex, but well documented APIs or more user-friendly formats. None of the tools allow creating rules on-the-fly in the code. Sensei does allow this, which greatly improves the speed and usability of writing rules.

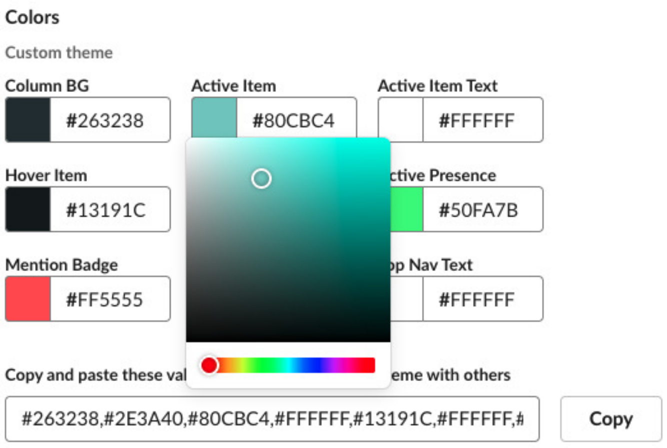
Recipe features

Because recipes can be created on-the-fly in the code, context-aware suggestions can be made, and testing of the recipes is more efficient since their markings can be observed live as the recipe is being created. This live preview in the recipe editor is mentioned by the security professionals as Sensei's most useful feature. Despite this, some of the features of the recipes and the recipe editor are not used as often or as effective as they could be.

When security professionals and developers create new recipes, they rarely use the code view. In fact, we have noticed that the code view can be overwhelming for some users who want to avoid learning a new language. To create recipes from scratch, and to adapt existing recipes, almost exclusively the UI view is used. The code view is only used by recipe-writers when copy-pasting recipes from the documentation or from recipes used as an example. The GUI can be updated to better reflect this behaviour. The UI view should be the main focus when opening the recipe editor, and the code view can be made smaller as its main focus is to copy-paste examples. The resulting GUI will have a similar user experience as the UI provided by Slack to customize (and share) color themes, as shown in Figure 9.1.

Security professionals report using the context-wizard to automatically generate recipes from context. However, in usability tests, none of the tested users have used this feature. This indicates that the feature might not be clearly understood. Instead of "create recipe for similar methodcalls", it might be more effective to make the option in the menu adapt to the context, for example "create recipe for Runtime.exec methodcalls".

Quick-fixes are added to nearly every recipe. However, in some situations no fully functional quick-fix can be created and the developer is still required to make changes to the code after applying the quick-fix. The template language that allows reusing parts of the original code is often required to create a working quick-fix. However, the suggestion box in the quick-fix menu is not clearly visible to the users, and as a result these suggestions are rarely used. The recipe-writer can still find them in the documentation or by copy-pasting from other rules, but using this menu should be more convenient. Currently, the variables are hidden by default and the "Show variables" button is not prominent


Colors

Custom theme

Column BG #263238

Active Item #80CBC4

Active Item Text #FFFFFF

Hover Item #13191C

ctive Presence #50FA7B

Mention Badge #FF5555

p Nav Text #FFFFFF

Copy and paste these val eme with others

#263238,#2E3A40,#80CBC4,#FFFFFF,#13191C,#FFFFFF,#

Figure 9.1: The theme editor in slack provides an intuitive UI interface on top to edit the theme, but also adds a code view and a copy button to allow fast and easy copy-pasting of existing themes.

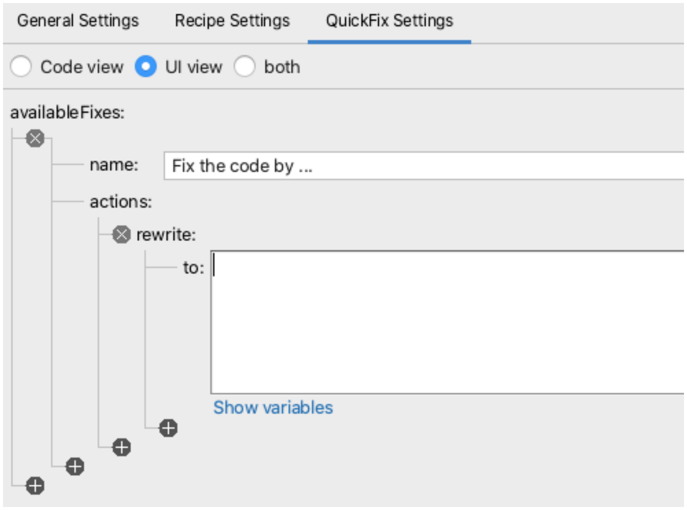
enough, as shown in Figure 9.2.

Scopes are used frequently in industry, almost exclusively to avoid creating EFPs and increase developer usability. Most of the comparable tools operate in later stages of the SDLC. They perform scans during code review or testing phases. It is often a security expert who will analyze and prioritize the results of security scans by placing them into the bug tracking system. Features for those tools often include integrations with common bug tracking systems to allow them to publish bugs automatically.

We observed that many tools provide functionality to disable certain rule reports through configuring security policies. This is a necessary feature to remove or hide classic false positives. However, this disabling of reports is designed to help security experts keep a good overview of the application state and to help prioritize more severe issues. With the exception of the Fortify Security Assistant that disables rules to speed up the scans, disabling rules themselves is rarely supported with the goal to improve the usability of the developers.

9.1.4 Feedback and remediation

Sensei is distributed as an IDE plugin. This allows it to reuse and extend existing IDE functionality, and hence feel like a natural extension of the developer's tool kit. When interviewed, subjects of the usability



[✓ UI view]

availableFixes:
name: Fix the code by ...
actions:
rewrite:
to:

Figure 9.2: The suggestions in the quick-fix menu are hidden by default. Users do not make use of the "Show variables" button that reveals them, as it does not attract their attention.

tests reported that the markings and quick-fixes felt similar to those provided by the IDE itself. They all indicated that they would use them frequently.

When analyzing newly written code, the longest time we have measured that is needed for the analyses to finish is 29 ms. This is far below the threshold of 125 ms to be considered real time. A developer is hence not hindered during development unless they are violating a coding guideline and need to use remediation.

When code is marked, the developer needs to spend some extra time understanding the issue and fixing it. During the empirical experiment of Section 8.1, the mean observed remediation time of a guideline violation is 19 seconds for users of the test group and 129 seconds for users of the control group.

On average, the use of Sensei increased the total development time with 11%. This is a relatively low increase considering the programming assignment was to complete security-critical features and hence the subjects were frequently confronted with feedback from the tool. It is also important to note that for all of the subjects, the experiment was the first time they were making use of the tool. In the control group, an average of 8.8 violations were left at the end of the assignment, whereas only 0.22 violations were remaining on average in the test group. If the subjects of the control group had addressed all remaining violations and maintained the mean remediation time, this group would have a mean development time 17% longer than that of the test group.

During the experiment, 98.4% of code markings shown by Sensei (n = 247) were resolved by the developers. Out of the resolved code markings, 73.3% were fixed using the quick-fixes. All of the users (n = 15) used at least one quick-fix, with an average of 12.71 (s = 4.73) quick-fixes used. The remaining code markings have been removed manually, either by fixing the violation or by removing the violating code entirely. This is a high level of engagement, compared to the lower than 20% "Apply fix" rate reported by the code review tool Tricorder [107]. On average the subjects resolved the issue within 19.10 s (s = 25.22 s) of writing the violating API call. Developers appear to be spending comparatively little time understanding the issues and applying fixes. By comparison, for Tricorder and SpotBugs the time between writing the violating code and fixing is usually several days [107, 116].

In the experiment with Sensei, only 1.6% of code markings were ignored, which is a low EFP rate. After carefully improving their analyzers, Tricorder reached an EFP rate of around 5%. Despite its great attention to developer usability, during an experiment with Application Security plugin for Integrated Development Environment (ASIDE), 63 of 101 (62%) markings were addressed [117]. When using SpotBugs, research reports that 58% of the found issues were never reviewed and out of the reviewed bugs, only 55% were eventually fixed [116]. Early Security Vulnerability Detector (ESVD), a tool with heavy focus on early detection of vulnerabilities, but without customized rules, found that the test group in their experiment only resolved 53% of the markings [118].

We observe a big gap in EFP rates with Sensei (1.6%) and Tricorder (5%) on one hand, and ASIDE (38%), ESVD (53%), and SpotBugs (77%) on the other hand. The reason for this is likely the customization of rules. Tricorder allows creating new analyzers and their quality is closely monitored. For Sensei, developers are given carefully tailored rules, often written by the engineers themselves, and relevant to their project. These efforts improve the usability of the tool and hence result in increased trust by the developers.

Impact on security

The usability measurements presented so far suggest that when Sensei is used, the secure coding guidelines are applied most of the time. During the first industry trial of the plugin, described in Section 8.3, the client has tracked closely whether or not the enforced guidelines actually prevented the introduction of vulnerabilities early on. The trial was done with five developers for the duration of three months. They reported a total of over 200 confirmed bugs being prevented. The most common issues involved sensitive information leakage and tapjacking vulnerabilities in their mobile application.

A limitation of the tool's local analyses is that they do not allow us to detect whether or not a certain input has already been sanitized before flowing into the routine being analyzed. This is in line with our approach and goal of enforcing coding guidelines that defend every routine for future use, i.e., such that it is still secure whenever it might be reused with unsanitized data. So if the local analyses identify a lack of local sanitization, the developer will be expected to let the routine sanitize that input again. At first sight, this might result in the same data being sanitized multiple times within an application, which will negatively impact performance.

In practice, however, this proves to be largely a non-issue. In practice, APIs are not designed in a vacuum. Instead they are developed with potential application architectures in mind. Furthermore, when concrete applications are first designed, security and application architects also take into account best practices for secure architectures (that is, if they care for security-by-design). Similarly, the coding guidelines can be co-designed with certain application architectures in mind. Doing so provides an easy mitigation of the potential issue of redundant, multiple sanitizations.

For example, consider the case of XSS attacks. It is a common misconception that in order to prevent stored XSS attacks, user input should be encoded before it is stored in the database. A better recommendation is to encode the database output when it is used, as the stored data may be used in different contexts, requiring different encoding methods. For example, a string value may be displayed on the HTML page and also used in a JavaScript script on that same page, resulting in two different, but simultaneous escape requirements. We learn that data should always be sanitized before it is stored in the database and encoded before it is displayed in the web or mobile application. Since these are usually the ends of the data flow, no data needs to be sanitized or encoded twice. If the rules are co-designed with the secure application architecture, encoding routines can be enforced only at the correct locations in code. The above does not imply that Sensei is the one and only tool that solves all potential software development security issues.

To detect issues as early as possible, i.e., in real-time as the developer is writing code, analyses have to be light-weight. This implies that all possible execution paths in the entire program cannot be exhaustively considered, and some types of vulnerabilities, including design flaws, can go undetected. This is a common trade-off, therefore tools that are used early in the SDLC such as Sensei should be complemented with more complete scanning solutions deployed later in the SDLC. Security professionals understand this. In the second industry trial Sensei is complimented by five additional security tools: Fortify, Checkmarx, SonarQube, Semgrep, and FindBugs.

An example of this strategy also exists with multiple products of the same company. The Fortify Security Assistant IDE plugin is used earlier in the SDLC than other Fortify tools, but only uses a subset of the available rules to improve developer usability. It helps detect a set of vulnerabilities earlier, and hence saves money and time fixing those issues, but it does not provide the full protection that, e.g., Fortify on Demand does. In the related work section, we will discuss where we consider Sensei to improve over Fortify Security Assistant as an early SDLC tool.

9.1.5 Project and team management

Compliance

Coding guidelines can provide a good measure for security in a software product. Where vulnerability scanning can only provide an indication of the vulnerability density, they do not provide the full picture. In the case, for example, where a large number of SQL injections is found, this could indicate poor database security. But it can also mean that there simply are a lot of database queries, with a large portion of them done securely. For coding guidelines a relative measure can be designed, by comparing the number of guideline violations to the number of times the code complies to guidelines. Since complying to strong coding guidelines leads to secure code [119, 120], we get a better indication of the security in the software product.

The plugin is useful as a tool to aid the developer, but the option to measure guideline deployment also hints at its potential as a management tool. As demonstrated in the experiments, management can track the changes developers makes to projects and log the guideline violations that they introduce and fix, with or without aid of the quick-fixes. Currently, this data is collected in the events databases on the machine of each developer. In the future, these metrics should be collected and visualised on the SCW platform. This makes the ROI clear to companies using the tool. With this data available in the platform, more targeted and individually tailored training can be provided as well. This data can for example feed into the ITS to improve its recommendations, as described in Section 5.2.2.

Integration in other stages of the SDLC

While individual performance can hence be measured and improved, with developers working in different branches, and hence different states of the project, it is hard to get a good overview that way. To resolve this, we also give managers the possibility to use the plugin technology as a headless scan that can be performed from the command line. However, in practice, we have noticed that the performance of this headless scan is lacking. In the IDE, recipes are verified against the code in the current file of the editor, the code the developer is working on. In the headless scan such context is not available, so every file in the project needs to be inspected, this slows down analyses. Security professionals also indicated that better integration with CICD tools is needed. This lack of automation in different stages of the SDLC is critical for the security team. The security professionals in the second industry trial have spent time and effort to recreate Sensei recipes in different tools in the SDLC to compensate for Sensei's lack of CICD integration. While the tool is a developer tool first, it is also a security tool, and it is usually purchased by the security team. Which is why it is important to show value for both user groups.

Roll-out

From experience, we learned that the plugin is ideally rolled out when new recipes do not mark any existing code. This is when a project kicks off and zero lines of code have been written. Alternatively it can be rolled out when a new API or library is introduced in the project and recipes will be written for this library or API. Few projects are developed from scratch, however, so the reality is that the plugin needs to work in an already developed product. In that case, rolling the plugin out with all recipes switched on can be overwhelming to developers, as they are presented with a huge number of violations. In addition, developers are often hesitant to fix issues they did not introduce in the code themselves, and they might not even have permission to change code that is not theirs. This results in a large number of EFPs, which we want to avoid.

When developers create their own recipes from scratch, they are working on a certain branch of the project. They usually create targeted recipes to fix or enforce small things in the project files they are working on. When they create the recipe, they inspect the violations and fix the markings. The recipe and fixed code are pushed to the codebase simultaneously. This typically leads to few EFPs. However, often the application security team of the company imposes recipes as well. At one point, the security expert at a client of ours created a large number of recipes and imposed them onto the developers without fixing any of the resulting violations. It did not come as a surprise that this resulted in a great number of EFPs and out of the 20 developers that had the tool installed, all of them had disabled its markings out of frustration. To avoid such failures, we recommend two approaches to keep the EFP rate low for imposed cookbooks.

Firstly, in the ideal scenario the security team creates a number of recipes and looks at their violations in the code to inspect their severity. When recipes result in few violations, the team can safely roll out the recipes without resulting in too many EFPs.

The roll-out is more challenging when a recipe results in a large number of violations that are not trivially resolved. In that case the security team should create a developer task force. Their task is to create APIs to resolve the recipe hits. They then turn the original recipe into a library adoption recipe and fix all marked code with this recipe. In the process of doing so, many corner cases can be encountered that help to fine-tune both the API and the recipe. The new API, the recipe, and all code fixes can be pushed to the codebase simultaneously.

The ideal scenario might not apply in practice, however. It is possible that the codebase is simply too large to start fixing all code markings. We have had clients where a strong recipe resulted in over 3000 violations. It can also be the case that when the security team creates a task force, this developer time is paid by the security budget, not the development budget. In such cases it is not beneficial to spend developer time to fix the existing issues in the code before rolling out the recipe.

We then instead recommend the second approach, in which the recipes are rolled out company-wide without fixing the code markings. In order to keep the EFP rate sufficiently low, the violations are only shown partly. For this purpose the option is added to the recipe editor, to only mark a recipe on newly developed code. This way only new violations are shown (and resolved) without resulting in an overly large EFP rate.

9.2 Perspectives

9.2.1 Improved recipe creation

Security professionals report that Sensei is the easiest tool they have used when it comes to creating new recipes. They attribute this mostly to the preview panels in the recipe editor, rather than the specific YAML syntax. Developers, on the other hand, are not used to creating rules for any tools, so they usually have nothing to compare it to. This is evident from the usability tests, where developers were more hesitant and more easily overwhelmed by the recipe editor compared to the security professionals. To reduce this hesitation, in the previous section, a design was proposed that would make the UI view the main focus in the recipe editor. In this design the code view would only be used for copy-pasting recipes. However, this still requires developers to create recipes for an analysis tool, a task they are not familiar with.

Instead, it might be possible to let developers create recipes simply by writing code. Currently, Sensei is able to apply a code transformation based on instructions from a recipe. In the future, it might be possible to do the opposite, and generate a Sensei recipe from a code transformation. If this technology exists, the recipe editor can simply show two code panels side by side. The left panel can be a static view of the current state of the code, while the right panel allows the developer to make (small) changes to the code. A Sensei recipe can then be created from the code changes that can optionally be adjusted in the next step.

This technology would also enable automatic recipe creation methods, such as generating a recipe from a code patch in the code repository. It might even be possible to dynamically suggest recipes while observing the developer during their normal workflow. Previous research has been performed to identify API rules for cryptography from code changes [121]. Efforts have also been made to automatically generate patches from code repositories and their histories, using different algorithms [122], including those learned from human-written patches [123] or correct code [124]. While this research tries to automatically patch bugs, the approaches can also be used to create recipes to apply the discovered patches more broadly and to do so during the writing of code rather than afterwards. With some user interaction, such a tool might also be able to generate recipes (without fixes) from the output of traditional security tools.

This technology is part of ongoing research funded by a Vlaams Agentschap Innoveren & Ondernemen *(English: Flanders Innovation &*

Entrepreneurship) (VLAIO) Onderzoek & Ontwikkeling *(English: Research & Development)* (O&O) project as of 2019.

9.2.2 Adapting feedback to the skill level

An important concept during the design and evaluation of the Sensei IDE plugin, is the EFP rate. When many markings exist in the code that the developer does not intend to fix, i.e., when there is a high EFP rate, this might cause developers to be overwhelmed and ignore feedback from Sensei altogether. To explain this concept, the example of OS command injection was used. A simple and easy to understand recipe to avoid OS command injection, is to simply avoid all uses of OS commands. A more experienced developer, however, will understand how OS commands can be used securely, for example to launch a different software application through a hard-coded command. This recipe will lead to an EFP for an experienced developer, but might be more easily understood by a developer with no security skills than a more advanced recipe.

In other cases, recipes are created that detect (presumably) deliberate insecure configurations. Take the example of cookies, where it is generally recommended to configure them as HttpOnly. This prevents the cookies from being used in client-side scripts, and hence avoids some of the most common XSS attacks. However, in some legitimate cases the developer might need to use a cookie in a client-side script, and to configure the cookie as such. Of course, they have to take the security implication of this configuration into consideration. For example, they will have to ensure that this is not used for security-sensitive cookies, such as session cookies. A recipe that detects insecure configurations like this, will lead to EFPs for developers who need the features that are blocked by these configurations.

Both the example of the OS command and cookie configuration, lead to EFPs for security experts, but are still important recipes to enforce for a novice developer. Fortunately, through integration with the SCW portal, a user ability estimate is available, such that the feedback for recipes like this can be adapted to the skill level of the developer. For developers with a low ability level, this recipe can be shown as an error, while the more experienced developer can be shown a warning or information level marking. The descriptions can also be adapted for each skill level. The less experienced developer can be shown the simple and easy to understand guideline, to use the most secure configuration. To a more skillfull developer it will be less overwhelming to explain the security implications of the configuration, and how to mitigate them in other parts of the code. Adapting UIs is most often based on the experience of the user with the interface itself rather than their knowledge in a specific field [125, 126]. This research indicates that optimizing UI design based on novice learning rather than long-term efficiency by experienced users can be counterproductive. It remains future work to assess whether or not a more optimised UI will be required for advanced Sensei users.

9.2.3 Controlled experiment in industry environment

The experiments described earlier in Part II of this book mostly contain second-hand information from industry trials, as well as a controlled experiment with students that focuses on validating some of the features contributing to the usability of Sensei. In contrast to the first part of this book, no strong empirical evidence is presented that validates the efficacy of a tool like Sensei, or the paved path methodology in general.

In this section, I discuss the design of future experiments that could be conducted to gather stronger evidence for the proposed solution.

Subjects

In an industry setting, it is not practical to divide a team's members into a control group and test group of similar skill level. Instead, it might be feasible to compare two teams and allow the use of Sensei by only one of the teams. Alternatively, the same team could first be observed for a duration of several days or weeks without use of Sensei and then Sensei can be introduced and results can be compared over time. These approaches can be applied to evaluate several research questions depending on the goal of the experiment. In the remainder of this section, the research goals and research questions of three such experiments are outlined.

Research questions

In the controlled empirical experiment of Section 8.1, the focus was on validating the effectiveness of several Sensei features at helping developers adhere to secure coding guidelines during development. The results showed that Sensei code markings were addressed often, and were addressed fast. Of the guidelines violations introduced by subjects of the test group, 98.4% were resolved with a mean remediation time of 19.10 s.

However, developers in an industry environment are likely to use Sensei differently to the subjects of this experiment. Not only because they are more experienced, and are likely more familiar with alternative tools, but also because they work in larger teams and with stricter deadlines.

Feature validation In one experiment that can be conducted in the future, the same features can be evaluated in an industry environment. The *goal* of this experiment would be to observe the impact of the Sensei plugin on the developers in an industry setting. The *purpose* would be to evaluate the usability and effectiveness of some of the features of Sensei. The *quality focus* is the ability of the plugin to help developers adhere to the secure coding guidelines that are in place and the impact on their cognitive burden while doing so.

The following research questions can be answered in this experiment:

- **Q1** What is the effective false positive rate in an industry environment?
- **Q2** What is the mean remediation time of markings in an industry environment?
- **Q3** Do developers in an industry environment often use the provided quick-fixes to resolve code markings?

This data could be analyzed and compared to the findings of the experiment with students. This could both evaluate the Sensei features more thoroughly as well as give us insights into the differences between developers in educational institutions and developers in industry environments. Experiments like this can help researchers and enterprises that design security tools improve the usability and effectiveness of their tools. However, to convince organizations to adopt role-specific tools more widely, we should evaluate their impact on speed and security during development.

Efficacy of the paved path methodology The *goal* of the experiment is to measure the impact of the paved path methodology on development and delivery of software. The *purpose* is to evaluate the ROI for organizations that consider using this approach. The *quality focus* is the impact of the methodology on the delivery speed as well as the security of the delivered code. In industry environments, a trade-off exists between fast delivery of new software features and the security of the software. Improving one at the detriment of the other is not real improvement. In this experiment, it is hence crucial that both these metrics are taken into account. By using the paved path methodology, we expect that security problems are addressed earlier in the SDLC, and this will positively impact the security of the code. We also expect that because of this early intervention, time will be saved in later stages of the SDLC as less markings will be found and subsequently need to be fixed. It is expected that remediating potential issues during development will require less time investment compared to remediating them in later stages, resulting in an overall increased speed of delivery.

The above goal can be achieved by means of an experiment aimed at answering the following questions:

- **Q1** Is the speed of delivery impacted meaningfully when developers use the paved path methodology?
- **Q2** Is the number of findings by security tools deployed later in the SDLC impacted meaningfully when developers use the paved path methodology?
- **Q3** Related to **Q2**, is the time and resources invested in finding and remediating vulnerabilities in later stages of the SDLC reduced when developers use the paved path methodology?

Delivery speed can be measured through several metrics [127].

- *Lead time for change* is the time it takes between a customer request and the request being satisfied.
- *Deployment frequency* is the frequency with which working software is deployed to production or distributed to the customers.
- *Change failure rate* is the frequency with which deployments require rollback or other measures to amend failures.
- *Time to restore service* is the time it takes to restore service after an incident or failure.

While the goal of this experiment is to evaluate the paved path methodology, this methodology is inherently tied to the use of role-specific developer tools. The evaluation is hence closely related to the efficacy of the tool itself and it is possible that the outcome of this experiment underestimates the efficacy of the paved path methodology due to imperfect design and implementation of the used tool.

Efficacy of Sensei in supporting the paved path methodology Finally, an experiment could be set up to evaluate to which extent Sensei helps development and security teams adopt practices in line with the paved path methodology. The *goal* of this experiment is to observe the impact of Sensei on the processes and interactions between security professionals and developers. The *purpose* is to evaluate if Sensei is effective at supporting the paved path methodology. The *quality focus* is on the interactions between different team members and the purpose of the Sensei recipes that are being developed.

The above goal can be achieved by means of an experiment aimed at answering the following questions:

- **Q1** Do security professionals frequently need to consult developers to create quick-fixes?
- **Q2** Do developers frequently write recipes for other purposes than security?

Part III
Closing

Chapter 10

Related work

Read them have you?
Page-turners they were not.

Yoda

Software security is a relatively new field [128], but many tools and practices have already been developed that have caused great advancements.

If nothing else, take away from this chapter...

Security begins even before code is written. Laws, legislations, and consumer demands all impact how much attention is given to security. Besides lightweight linter tools, developers can also find help to produce secure code from patterns, libraries, and frameworks. In the build phase, the use of new methodologies has driven the automation of building executables and installing dependencies, which has made it easier to test for use of vulnerable components.
Most security practices, however, take place in the test phase. Many code review practices and tools exist, most of which allow customization of the rules through one of three methods: an API, a custom query language, or a formatting language such as XML or YAML. Finally, in the release phase, recent advancements in Infrastructure as Code (IaC) have made it easier to securely deploy applications and manage infrastructure.

Many guides exist to help you decide which tools are appropriate for your project. In this chapter, I want to extend these guides so that you are better equipped to estimate the strengths of a tool for use in the paved path methodology. I describe commonalities between tools in different phases of the SDLC and how they can be deployed effectively. For future reference, Appendix F contains a number of battlecards with my thoughts on a few more tools.

10.1 Governance

Considering security begins before any code is written. Priorities set by management and the business itself can have a big impact on the security practices deployed during the SDLC.

10.1.1 Training

Training has always played a critical role in software development, because standard computer science and engineering education often neglects software security.

Companies should first offer security awareness training to all employees involved in the SDLC. Security awareness training does not necessarily need to be tailored to a specific audience. Developers, Quality Assurance (QA) engineers, project managers, and operators can all partake in the same training. A generic introductory course like this however is insufficient, the next step is to provide role-specific individual training. As explained in this work, developers should be taught secure coding, and not follow training intended for security professionals or penetration testers.

In the ideal scenario, a company should also verify or provide training for vendors and contractors. They should require annual refreshers for all employees and can host software security events to nurture a good security culture.

10.1.2 Compliance and policy

Software security is not only a problem of enablement. Good enough training, tools, and processes exist today that can embed security in software development from the start. Yet, we still see frequent reports in the media of bad software practices and vulnerabilities that easily could have been prevented. The reality is that businesses often prioritize getting to market and getting features out, over their obligations in terms of security. Not enough incentives are in place for businesses to put more emphasis on the security of their products. Too often it is simply considered an afterthought and a necessary cost.

Privacy and trust

The resulting security problems, however, do not only hurt the business, they also hurt the consumer. When businesses are hacked, it is often private data of consumers that is leaked. Recently, consumers have claimed control and rights over their personal data. Legal frameworks have been built around data privacy, forcing businesses to consider data protection more seriously.

Most famously the GDPR, a law on data protection and privacy, is enforced in the European Union (EU) since May 25, 2018 [129]. It contains regulations that strengthen the individual's fundamental rights in the digital age and clarify rules for businesses storing or processing personal data of individuals in the EU. This law forces businesses to consider security more seriously, as it is estimated that at least 25% of software vulnerabilities have GDPR implications [130]. Non-compliance with the general data processing principles in this law can result in significant fines, for example in June 2021, Amazon was fined €746 million[1]. Two years after the implementation of the GDPR, the European Commission (EC) found that individuals' knowledge about data privacy has increased, and as a result privacy has become a competitive quality for companies which consumers are taking into account in their decision-making [131].

Security is no longer just a necessary cost during development, but businesses are able to see a more direct ROI for high quality software security. Businesses put more effort into the appearance of having trustworthy data protections in place, a process called trust management [132, 133]. In this discipline, consumer trust is the end-goal and good security practices are a means to this end. To convince consumers and buyers of software to trust a product, businesses can acquire a seal of approval from a third party to prove they adhere to certain standards. One such widely known certification is the International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) 27000-series, or ISO27K for short. This series provides best practice recommendations on information security management, covering privacy, confidentiality, and other cybersecurity issues [134]. Other regimens that companies often aim to comply with are Payment

[1] `https://www.enforcementtracker.com/ETid-778`

Card Industry Data Security Standard (PCI DSS) and Health Insurance Portability and Accountability Act (HIPAA). To add transparency, businesses can also provide a Software Bill of Materials (SBOM), that lists all components used in their software [135]. Such an SBOM is most easily produced using build tools as explained in Section 10.3.

In the United States (US) Biden Executive Order (EO) on Improving the Nation's Cybersecurity issued May 12, 2021 the National Institute of Standards and Technology (NIST) was ordered to publish guidelines regarding practices that enhance the security of the software supply chain. Besides providing the purchaser with such a SBOM, there are numerous other standards and procedures listed regarding trust, multifactor authentication, encryption, and use of automated security tools. In this EO, NIST is directed to solicit input from the private sector and academia to develop standards, tools, and best practices. Among the more than 150 position papers, dr. Matias Madou, dr. Brian Chess, and I have also submitted two. One position paper advises the creation of a certification framework for education in secure development practices. The second promotes the use of the paved path methodology. Both papers have been accepted and published on the NIST website [24]. Time will tell if this EO makes as big of an impact as the GDPR.

Besides the GDPR and the US EO, there are of course similar laws in other parts of the world such as the Data Protection Act in the United Kingdom, the Privacy Act in Canada, and the Personal Data Protection Bill in India.

Law enforcement access

No discussion on laws and data privacy would be complete without mentioning laws on the collection and storage of electronic communication and their access by authorities. Many of these laws contain requirements that force operators of end-to-end encrypted systems to undermine this encryption, so that law enforcement can be provided access to user communications. One such example is the draft considered by the Belgian government at the end of September 2021[2]. Under this law, operators would have to be able to "turn off" encryption for specific users, essentially creating so-called backdoor access. The consensus among cybersecurity experts is that there is no way to provide third-party access like this to end-to-end encrypted communications, without also creating encryption backdoors and vulnerabilities that can be exploited by malicious third parties [136]. Creating a backdoor like this, undermines the

[2]`https://ibpt.be/index.php/operateurs/publication/annexe-1-dispositif`

whole security of the system and puts its users at risk [137].

In other countries where similar legislations have passed, such as Australia, research has shown that this has discouraged companies from offering new end-to-end encrypted products [138]. It is safe to say that policy makers and governments can have a significant influence on the security of software products, for better or for worse.

10.2 Develop

Developer toolkits evolve over time, and many new technologies and frameworks exist to help developers produce code more efficiently, and more securely. As explained in Section 6.1, security tools are handed to the developer as well because a shift left movement is ongoing to try and identify possible security problems as early as possible in the SDLC. These tools, however, still use a reactive testing-based approach and can usually only identify security problems once sufficient code has been developed. These tools will be discussed in the section on testing, Section 10.4. In this section, I discuss tools and practices that help a developer produce secure code from the start.

10.2.1 Lint

Linter tools are designed to allow the developer to concentrate solely on the algorithms, data structures, and correctness of the program, and only later, with the aid of lint, address non-functional aspects of the code. They mostly focus on syntax and styleguide checking but some tools are advanced enough to check for certain bugs as well. Depending on their targets, linters perform their analyses with string-matching or reduced versions of ASTs without symbol information. The more advanced lint tools perform similar analyses to Sensei, making use of the entire AST. Lint tools are useful and commonly used, but they are not often deployed for security purposes. Some examples of lint tools are Error Prone[3], Checkstyle[4], PMD[5], and SonarLint[6] (Appendix F, battlecard 1) by SonarSource. Lint tools are also often included by default in IDEs such as AndroidStudio[7] and IntelliJ IDEA[8]. Not many secu-

[3]http://errorprone.info/

[4]http://checkstyle.sourceforge.net/

[5]https://pmd.github.io/

[6]https://www.sonarlint.org/

[7]https://developer.android.com/studio/write/lint

[8]https://www.jetbrains.com/help/idea/code-inspection.html

rity rules are included in lint tools by default. Out of the tools above, SonarLint supports the most, with 29 rules targeting vulnerabilities in Java[9]. This is because lint tools require fast response times, and scanning for vulnerabilities often takes longer-running analyses. Many lint tools are open-source which means their rules can be customized to enforce secure coding guidelines, but none are designed for easy and fast customization of the rules.

10.2.2 Security patterns

Research has shown that adherence to secure coding guidelines leads to more secure code [79, 139]. It comes as no surprise that many efforts exist both in industry and in research to develop such guidelines. In contrast with vulnerability lists, discussed in Section 10.4, these patterns provide proactive guidelines targeting developers. Many of these guidelines can be used as a basis to create Sensei recipes, or rules for similar tools, provided they are specific enough.

Some provide sufficiently clear API-level instructions that can directly be implemented as recipes in our plugin. We have demonstrated this by creating a rule set from The Android Application Secure Design/Secure Coding Guidebook by the Japan Smartphone Security Association [140]. Other notable examples are the guidelines designed to counter side-channel attacks, designed by Witteman [141], and the Oracle Coding Standards[10]. The Java code issues and transformations in these guidelines fall clearly within the capabilities of Sensei.

Other guidelines are too generic and high level such as the work by Schumacher et al. [142], or the OWASP Proactive Controls[11]. In order to support these with Sensei or other tools, they need to be translated into concrete guidelines and customized for the used APIs. For example, the OWASP Proactive Control number 5 instructs to validate all inputs. To apply this proactive control in practice, security libraries have to be developed or selected to perform the input validations.

Some efforts have also been made to automatically generate rules from code changes such as Paletov et al. [121]. As mentioned in Section 9.2.1, in the future we also want to develop such automatic recipe creation methods.

[9]https://rules.sonarsource.com/java/type/Vulnerability
[10]https://wiki.sei.cmu.edu/confluence/display/java
[11]https://owasp.org/www-project-proactive-controls/

10.2.3 Security libraries and frameworks

Another solution to make developers adhere to coding guidelines, is to implement them into frameworks or libraries. An example is the OWASP ESAPI, an open-source application security control library that provides clear replacement APIs for insecure JDK implementations [106]. As mentioned in this work, Sensei recipes have already been developed to support replacing banned methods with alternatives from the ESAPI as a demonstration of library adoption recipes.

Popular web application frameworks provide methods for sanitizing inputs and escaping outputs to prevent common vulnerabilities. These frameworks create a paved path for developers to follow. We have observed that these efforts result in useful code examples in the documentation of frameworks that are easy to understand for developers. They also make for easy development of Sensei recipes to adhere to these guidelines. However, the implementation details of these methods are sometimes lost to developers, and the results from this work show that this can sometimes lead to increased difficulty locating vulnerabilities.

Nonetheless, this evolution in frameworks has shown to be effective at preventing security problems as indicated by the position of injection flaws and XSS in the OWASP top 10. Despite XSS attempts remaining common [1], the vulnerability has moved from third place in 2013, to seventh place in 2017. In 2021 it is likely to merge with injection flaws, as shown in Figure 5.1 on page 93. After being the top category since 2013, injection flaws are likely to move down to the third position in the OWASP top 10 2021.

10.2.4 Artificial intelligence code completion

Recently some tools have emerged to help developers produce code more efficiently with the help of Artificial Intelligence (AI). Some examples of AI code completion tools are Copilot[12], tabnine[13] (formerly codota[14]), and kite[15]. The most famous, Copilot, is a service created by GitHub and OpenAI described as “Your AI pair programmer”. It functions as a plugin for Visual Studio Code that generates code based on the current file formats.

Copilot is trained on public code and text from the internet, including public repositories on GitHub. It uses context from the IDE to

[12]`https://copilot.github.com/`

[13]`https://www.tabnine.com/`

[14]`https://www.codota.com/`

[15]`https://www.kite.com/`

provide suggestions in the code that is being developed.

Many developers that used Copilot found that it was accurate, and often improved their productivity. Generally, when developers struggle with the implementation of a feature, they will often Google it. In many cases they will either find documentation or a Stack Overflow post that will point them to a solution. It still requires judgement from the developer to decide if the solution is correct and secure.

Similarly, the code suggestions Copilot provides are only as secure as the repositories it was trained on. Copilot was used to complete 1,692 code scenarios with risk of introducing vulnerabilities. Researchers found that in 40% of cases the suggested code was vulnerable [143]. It is clear that it the developer's judgment is still required to ensure the produced code is secure.

10.3 Build

There is an evolution in software development towards increasingly iterative and feedback-driven strategies. Most noticeable is the Agile development model, formally introduced in 2001, where customer collaboration and responsiveness to change are key components [144]. The highest priority in this model is to satisfy the customer through continuous delivery of valuable software, by welcoming changing requirements, even late in the development process. Working software has to be delivered frequently, in a couple of weeks to a couple of months. Product management and developers have to work closely together to set priorities and iteratively deliver minimal viable products and improvements. In this process, individuals and interactions are prioritized over processes and tools, and working software is prioritized over comprehensive documentation. The Scrum framework is frequently used to implement this type of development strategy [145]. Generally, security benefits from things that hold still. When the codebase remains the same for longer, the security team has more time to test and fix the security of the code. The increased rate of change has made the job of the security professional more challenging.

10.3.1 Build tools

Building on agile practices, DevOps aims for complete end-to-end automation of not only software development, but also delivery. In academic research, there is not yet a clear definition for DevOps, but it is most often characterized by cross-functional teams and shared ownership [146, 147]. Quality deliveries with short release cycles need a high degree of automation, and many tools have been developed to assist with this automation.

Build tools are used for compiling code, they often include so-called package or dependency managers to centralize project dependencies. Some examples of build tools and package managers are Ant[16], Maven[17], Gradle[18], Pip [19], and Yarn[20].

Managing dependencies centrally like this, makes it easy to monitor and update them to newer versions. This also provides a centralized overview of all software components used to create an SBOM as explained in Section 10.1.

10.3.2 Software composition analysis

Use of vulnerable and outdated components is a common vulnerability category, and part of the OWASP top 10. Many Software Component Analysis (SCA) tools exist that scan build files and alert developers when any of the used dependencies contain vulnerabilities.

Some notable tools are Snyk Open Source (battlecard 2), Dependabot (battlecard 4) and GitLab Dependency Scanner (battlecard 5). These tools are typically integrated into the code repository and run regular scans. Use of vulnerable components is a vulnerability that can be introduced *after* initial development because dependencies are (supposed to be) updated frequently. It makes sense to integrate this type of security tool in the code repository rather than development tools. In the development tool, many developers would get notified of an outdated dependency at the same time, while likely few of them would be working on the build file. This would either result in many EFPs, or in the same fix being applied by multiple developers. Remediation for these vulnerabilities is often simply bumping the dependency to the newest version, and results from the 2PL model show that developers have no difficulty fixing this type of vulnerability. As remediation guidance, SCA tools often create automated pull requests that update dependencies to a secure version. This process is intuitive and well integrated with existing developer workflows.

However, some challenges remain in this field, as in some programming languages over 70% of vulnerabilities are in transitive dependen-

[16]`https://ant.apache.org/`

[17]`https://maven.apache.org/`

[18]`https://gradle.org/`

[19]`https://pypi.org/project/pip/`

[20]`https://classic.yarnpkg.com/en/`

cies [148]. Transitive dependencies can not be easily updated since they are not in direct control of the developer. With some package managers (such as Maven) it is possible to exclude a transitive dependency and manually download the newest version. This comes with the risk of run-time errors if the newest version contains any breaking changes, since the developer has no control over the code in the direct dependency where these breaking changes may cause errors. It can also suffice to verify that the methods containing vulnerabilities are not used in the code. These methods can also be excluded or replaced with so-called monkey-patches[21]. All these options require more intimate knowledge of the package manager or the dependencies being used and make fixing this vulnerability type more complex than it is often represented in training.

10.4 Test

Software security initially started as part of software testing [99]. Today, still, most security practices are deployed in the testing phase. Some part of software security will always be reactive. Like all other parts of computer science, security keeps advancing, and we will always know more tomorrow than we know today.

So while many novel security tools and practices have been introduced and proven to be effective, new practices generally do not replace old ones. Instead, they are added to the arsenal of weapons that is available for development and security teams. In this section, I describe new tools and practices as well as some traditional ones, as I believe they will remain relevant, even if new tools and practices are being introduced.

10.4.1 Penetration testing

Penetration testing is the practice of breaking into running software by attacking it. Sometimes, the penetration tester has access to the source code to speed up this process. It is a common practice used by many companies and usually external experts are hired to perform these tests [5, 149]. Since the penetration tester needs access to the running software this can only be done late in the SDLC. Already in the introduction, we addressed that relying on security experts does not scale well. Furthermore, it does not integrate well in modern development strategies, where fast feedback cycles and frequent releases are key [150].

[21] `https://docs.plone.org/appendices/glossary.html#term-Monkey-patch`

Penetration testing does improve the security awareness of the developers, but does not cause any long-lasting change in development practices by itself [151].

10.4.2 Code reviews

To develop new features or fix bugs, a developer starts from a copy of the current codebase. As other developers submit changed code, this copy gradually ceases to reflect the main (or master) version. The longer development continues, the greater the risk of conflicts when merging work back into the main version. A code review is a manual inspection of produced code that is performed when this work is merged back. It is usually done by another developer than the original author but with that author present. Code reviews have a clear positive impact on the presence of vulnerabilities [79]. They also provide an educational aspect for the developer whose code is reviewed [152]. The downside is that, similarly to penetration testing, it relies on internal or external experts and hence does not scale well.

Continuous Integration (CI) tools are developed to automatically build and review code as frequently as possible when the working copies of developers are merged into a shared main version. A build server is usually set up for this purpose, which will build and test the code after every commit and report the results back to the developers. This testing is done with automated tools, such as static analysis tools.

10.4.3 Static analysis

Static analysis tools, often called Static Application Security Testing (SAST) tools, are well researched [153–155] and commonly used to detect vulnerabilities [4, 5, 149]. Most tools can run automatically, and are easily adapted in modern development strategies as a result. Static analysis tools vary from robust and time-consuming analyses such as Fortify [156] (battlecard 13-14) and Veracode (battlecard 16) to light real-time analyses [157]. Several resources exist to help compare different tools, such as the OWASP list of source code analysis tools[22] and list of vulnerability scanning tools[23], as well as Kompar[24] which allows easy comparison between static analysis tools. In controlled experiments, static analysis tools proved to be more effective than penetration testing [158].

[22] `https://owasp.org/www-community/Source_Code_Analysis_Tools`
[23] `https://owasp.org/www-community/Vulnerability_Scanning_Tools`
[24] `https://kompar.tools/`

As explained in this work, frequent testing is useful, but analyses of traditional security tools often run too long to be well-integrated in developer workflows. These tools are often seen as a big inhibitor for the developer's productivity. To mitigate this, a shift left movement is ongoing to apply them as early as possible in the SDLC.

However, static analysis tools require sufficient code to be completed in order to detect vulnerabilities, and hence can usually not be used in a proactive manner. By customizing the rules, some tools can be tailored to ignore context, which can speed up their analyses even if they were not designed with this in mind. But even if local versions of the analyses perform well, they often do not provide fixes to resolve the discovered issues and hence do not enforce a paved path. They cannot be applied in a pro-active manner, like Sensei can. Examples of tools that provide fixes in the code review stage are Tricorder [107] (battlecard 11), its open-source version Shipshape (battlecard 12), Semgrep (battlecard 9) or Reshift Security[25]. Research with Tricorder showed that most developers go back to their IDE rather than use the code review tool to resolve the issues [107].

10.4.4 IDE-based static analysis

Because developers prefer to remediate problems in their IDE, and as part of the shift left movement, many static analysis tools are now available as IDE plugins as well. For some tools no effort has been made to adapt them to better suit the developer. They still perform identical analyses to the original tool, either remotely or locally. Examples of plugins like this are FindBugs (battlecard 6) and Spotbugs (battlecard 7). Some tools prevent the developer from making changes to the code while the scan is in progress. This is the case for the Fortify Security Assistant (battlecard 15).

Performing the full scan, however, often takes too long and as a result these tools are not very usable. In an attempt to be more developer-friendly, other tools provide lightweight versions of the analyses, such as Fortify Security Assistant (FSA) (battlecard 15) as a lightweight version of Fortify on Demand (FOD) (battlecard 14), or Veracode Greenlight (battlecard 17) as a lightweight version of Veracode Static Analysis (battlecard 16).

In this case, the plugin is often not able to detect the complete set of vulnerabilities that the original tool is capable of. The goal is to provide faster feedback loops to the developer, with the drawback that

[25] `https://www.reshiftsecurity.com/`

```
Cipher.getInstance("DES");
```

Listing 10.1: Insecure use of a deprecated cryptographic algorithm

some of the vulnerabilities will go unnoticed. But by detecting a portion of the vulnerabilities earlier in the SDLC, they become easier and less expensive to fix. All the other vulnerabilities are still caught when the fully capable tool is used in later phases of the SDLC

10.4.5 Rule customization

For these tools to be most effective, their rules should be tailored specifically to the organization's coding standards and target vulnerabilities relevant to the organization [4, 5]. As described in this work, this also prevents false positives and EFPs, thus improving usability for developers and ensuring continued use of the tool.

Some tools do not allow customization of the rules, such as Reshift security, Snyk Code (battlecard 3), and Veracode (battlecard 16,17). Among the tools that do, different approaches exist. In this section, these approaches will be demonstrated with a rule to detect the use of the deprecated cryptographic algorithm Data Encryption Standard (DES) in Java. The insecure line of code that needs to be marked is shown in Listing 10.1.

API

The first method of rule customization is through use of an API. These tools require the rule-writer to write code that extends the functionality of tool so that it performs additional analyses. The tool, or sometimes only the extension itself, then needs to be built into a new executable that can be used to analyse software products. For Shipshape it is even required to expose this executable as a service using a docker image[26].

Creating detectors through an API allows for sufficient flexibility, but makes it more complex to develop and test custom rules. SpotBugs (battlecard 7) is an example of a tool that uses an API for rule customization by creating so-called third party "detectors" [159]. These detectors have to be implemented and compiled into a SpotBugs plugin. FindSecBugs [160] is a popular security plugin for SpotBugs. A detector to mark use of the DES algorithm is already implemented by the FindSecBugs plugin. In Listing 10.2, a snippet of the class `DesUsageDetector`

[26]`https://github.com/google/shipshape`

```java
public class DesUsageDetector extends CipherDetector {
    ...
    @Override
    int getCipherPriority(String cipher) {
        cipher = cipher.toLowerCase();
        if (cipher.equals("des") || cipher.startsWith("des/")) {
            return Priorities.NORMAL_PRIORITY;
        }
        return Priorities.IGNORE_PRIORITY;
    }
    ...
}
```

Listing 10.2: Rule customization of SpotBugs is done through java code using their API.

is shown that implements this detector. This code is copied from the `find-sec-bugs` project on GitHub[27]. The class extends the abstract class `CipherDetector` that is also implemented by the plugin, and hence not an API provided by the original tool. To create a detector for such a relatively simple example, multiple files and many lines of code are already needed that require sufficient knowledge of the APIs. While the creation of additional detectors is not as convenient as creating recipes with Sensei, at least the distribution of detectors through a plugin is convenient for users of the tool.

Other examples of tools that use APIs for rule customization are JavaParser[28], Tricorder (battlecard 11), Shipshape (battlecard 12), and Ruleguard (battlecard 18).

Custom query language

To make customization of rules easier, some tools provide a custom query language that makes abstractions of their API. They still require the rule-writer to write code, but they usually provide more specific syntax to make the development of rules easier.

Code Query Language (CodeQL) is an example of such a query language. It is a free and open-source semantic code analysis engine that is created with the goal to query code as if it were data. It borrows syntactic elements from data query languages such as SQL as well as elements from Java. CodeQL is used by the popular analysis platform Semmle (battlecard 8). A CodeQL query that detects use of DES is shown in

[27] `https://github.com/find-sec-bugs/find-sec-bugs`
[28] `http://javaparser.org/`

```
import java
from MethodAccess call, Method method
where
  call.getMethod() = method and
  method.hasName("getInstance") and
  method.getDeclaringType().hasQualifiedName("javax.crypto", "Cipher")
      and
  method.getParameter(0).toString().regexpMatch("DES.*")
select call
```

Listing 10.3: CodeQL query used by Semmle to find use of insecure algorithm DES.

```
result = All.FindByName("*getInstance*",11,11);
result = result.FindByParameterValue(0,"DES",BinaryOperator.
    IdentityEquality);
```

Listing 10.4: CxQuery query used by Checkmarx to find use of insecure algorithm DES.

Listing 10.3. This query is less complex and easier to write compared to using an actual API. CodeQL provides an online "Query console" that makes development of these queries easier. It provides syntax highlighting and auto-completion. With this console it is also possible to test the queries on several demo projects. The available projects do not necessarily contain the code a rule-writer is targeting, as was the case for the example rule. None of the 7 available projects is using the DES encryption algorithm.

The query language CxQuery by Checkmarx (battlecard 10) uses regular Java syntax. It provides abstractions to iteratively filter results based on certain properties of the code. Some knowledge of the API is required, but Checkmarx provides clear and easy to understand documentation that contains lots of examples. The resulting query, shown in Listing 10.4, is even shorter than the one for CodeQL and just as easy to understand.

Other tools that are using a custom query language include jQAssistant[29], and Neo4J[30].

[29] https://jqassistant.org/
[30] https://neo4j.com/

```
<Match>
 <QualifiedName>javax.crypto.Cipher</QualifiedName>
 <Method>getInstance</Method>
 <Arguments>
  <Argument>
   <Index>0<Index>
   <Value>
    <ComparatorOperator>equals</ComparatorOperator>
    <ExpectedValue>DES</ExpectedValue>
    <ComparatorType>String</ComparatorType>
   </Value>
  <Argument>
 </Arguments>
</Match>
```

Listing 10.5: SecureAssist rule to discover use of DES

Markup language

Some tools allow customization of the rules through markup languages such XML and YAML. SecureAssist [161] (battlecard 21) is an example of such a tool [162]. Additional rules can be added through a rule pack configurator. Rules themselves are written in XML format and the syntax is user-friendly and easily readable [163].

In Listing 10.5, an example rule is shown to discover uses of DES, as a comparison the same rule for the Sensei plugin is shown in Listing 10.6. SecureAssist's rule syntax is similar to Sensei rules. However, creating the rules requires learning their exact syntax, as no editor is provided, rules are created using any text-editor. Sensei, on the other hand, provides a rule wizard, context-aware suggestions, and a GUI to edit the rules as well as live-previews in the IDE. Sensei rules also support more comprehensive features such as the concept of untrusted variables and support for libraries.

Like Sensei, Semgrep (battlecard 9) uses YAML and its rule format is also similar to that of Sensei. While it is intended to be used as a testing tool, third-party plugins have been developed to use Semgrep in the IDE, potentially making it a great tool for supporting the paved path methodology. A rule to detect and fix use of DES is shown in Listing 10.7. It is important to note that Semgrep is the first tool discussed in this section that provides quick-fixes. Besides the search pattern and the fix, the rule also includes metadata that is stored separately for Sensei recipes, such as the category, severity level, and descriptions. Semgrep rules have a few advanced features, such as the concept of metavariables,

```yaml
search:
 methodcall:
  type: javax.crypto.Cipher
  name: getInstance
  args:
   1:
    type: java.lang.String
    value: "DES"
availableFixes:
- name: "Change to use AES/GCM/NoPadding"
  actions:
  - rewrite:
     to: '{{qualifier}}.getInstance("AES/GCM/NoPadding")'
```

Listing 10.6: Sensei recipe to discover use of DES

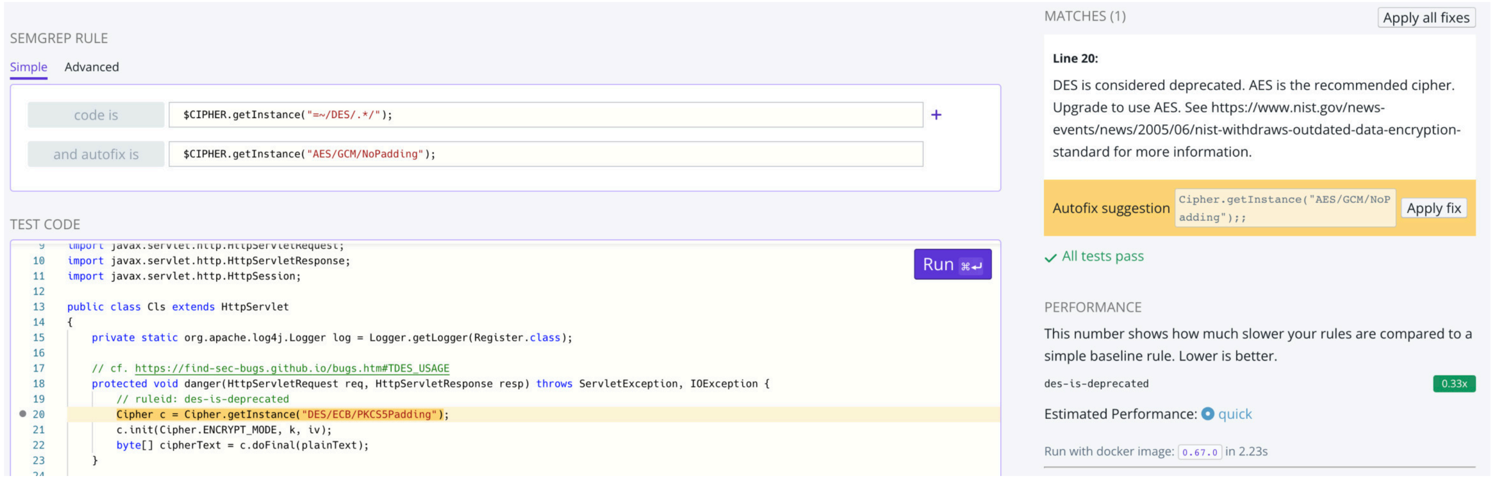
```yaml
rules:
- fix: $CIPHER.getInstance("AES/GCM/NoPadding");
  id: des-is-deprecated
  languages:
   - java
  message: DES is considered deprecated. AES is the recommended cipher.
  metadata:
   category: security
   cwe: "CWE-326: Inadequate Encryption Strength"
   license: Commons Clause License Condition v1.0[LGPL-2.1-only]
   owasp: "A3: Sensitive Data Exposure"
  pattern: $CIPHER.getInstance("=~/DES/.*/");
  severity: WARNING
```

Listing 10.7: Semgrep rule to discover use of DES

an abstraction made available to track variables such as method names across the search pattern. The metavariable `$CIPHER` is used in the example to define the fix. This is more user-friendly than the templating language provided by Sensei where the rule-writer is required to know the appropriate name of the element in the AST.

Semgrep provides a "Playground" on their website that allows recipe-writers to develop and test rules. In this editor, a rule-writer can use the "Advanced" view which is similar to Sensei's code view. A "Simple" view is available as well, but it does not provide the same level of support as Sensei's UI view does. As shown in Figure 10.1, this view still requires the user to know and understand most of the YAML syntax. Testing is unfortunately not real-time, but is completed in several seconds.

Semgrep also provides a few advanced features such as taint track-



SEMGREP RULE

Simple Advanced

code is `$CIPHER.getInstance("=~/DES/.*/");`

and autofix is `$CIPHER.getInstance("AES/GCM/NoPadding");`

TEST CODE

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class Cls extends HttpServlet
{
    private static org.apache.log4j.Logger log = Logger.getLogger(Register.class);

    // cf. https://find-sec-bugs.github.io/bugs.htm#TDES_USAGE
    protected void danger(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // ruleid: des-is-deprecated
        Cipher c = Cipher.getInstance("DES/ECB/PKCS5Padding");
        c.init(Cipher.ENCRYPT_MODE, k, iv);
        byte[] cipherText = c.doFinal(plainText);
    }
```

MATCHES (1)

Line 20:

DES is considered deprecated. AES is the recommended cipher. Upgrade to use AES. See https://www.nist.gov/news-events/news/2005/06/nist-withdraws-outdated-data-encryption-standard for more information.

Autofix suggestion `Cipher.getInstance("AES/GCM/NoPadding");;`

✓ All tests pass

PERFORMANCE

This number shows how much slower your rules are compared to a simple baseline rule. Lower is better.

des-is-deprecated 0.33x

Estimated Performance: ● quick

Run with docker image: 0.67.0 in 2.23s

Figure 10.1: Semgrep's "Simple" view in the Playground rule editor still requires use of the YAML syntax.

ing which is similar to the concept of trusted input in Sensei. To use taint tracking in a rule, sources and sinks need to be defined as well as optional sanitizers. Since taint tracking requires a source to be specified, it functions differently to the trusted input of Sensei, where all input is untrusted by default. Using metavariables it is possible to create a rule that prevents EFPs similarly to Sensei's trusted input. Listing 10.8 shows a rule to detect potential OS command injections, the analogous Sensei recipe is shown in Listing 7.4, but repeated here in Listing 10.9 for convenience.

```yaml
rules:
- id: os-command
  patterns:
    - pattern: (Runtime $RUNTIME).exec($COMMAND)
    - pattern-not: $RUNTIME.exec("$HARDCODED")
    - pattern-not-inside: |
        $COMMAND = getSafeCommand();
        ...
        $RUNTIME.exec($COMMAND);
  message: OS Command test
  languages: [java]
  severity: ERROR
```

Listing 10.8: Semgrep rule to detect OS Command Injection. Any commands passed on to the `exec` method that have not been retrieved through `getSafeCommand` will be marked.

```yaml
search:
  methodcall:
    name: "exec"
    type: "java.lang.Runtime"
    args:
      1:
        type: "java.lang.String"
        containsUntrustedInput: true
        trustedSources:
        - methodcall:
            name: "getSafeCommand"
```

Listing 10.9: Sensei recipe to detect OS command injection. Any input is untrusted by default except input retrieved through `getSafeCommand`. Untrusted input passed on to the `exec` methodcall will be marked.

Other tools that use formatting languages to customize rules are Fortify (battlecard 13-15), OpenRewrite (battlecard 19), ASIDE (battlecard 20), and GoKart (battlecard 22).

10.5 Release and deploy

High velocity of development and delivery is most easily achieved in Software as a Service (SaaS) and other cloud computing delivery models. It is easier to push frequent updates if there is only a single version of the application running, hosted centrally and managed by the software provider. Furthermore, because the service provider has access to user data and behaviour, it is easier to collect feedback and make incremental improvements. Finally, it is more economically viable to adapt to continuously changing requirements of customers, if the software is sold on a subscription basis.

10.5.1 Infrastructure as code

To keep pace with this high velocity of software development, new technology has been developed to automate infrastructure and deployment. With IaC, the process of managing and provisioning data centers is done through machine-readable configuration files rather than hardware configurations and interactive tools [164]. Most frequently these configuration files are declarative, focusing on what the eventual target configuration should be, rather than describing the necessary changes to meet this configuration. Two big components are required for automated infrastructure, those are application deployment and runtime orchestration.

Application deployment

Modern software applications often consist of a variety of services, such as an API, a web front-end, a back-end application, logging services, and services used for data analytics. To ease the deployment, and to isolate services from each other, virtualization is used. In early virtualization, a Virtual Machine Image (VMI) was created that contains the service's code and any requirements to run it, such as the OS and the dependencies. A VMI is a form of hardware virtualization, each is deployed as a *guest* on a *host* machine, providing its own OS with its own kernel. Because of this, a Virtual Machine (VM) can be deployed anywhere without requiring modifications to it. This also has the added benefit of isolation between different services, so that each one has a fixed amount of Central Processing Unit (CPU) processing power and memory. However, they are large and take a lot of resources to store and run.

```
# syntax=docker/dockerfile:1
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Listing 10.10: Example of a Dockerfile to build and run a Python application.

More modern technology moves from hardware virtualization to OS-level virtualization. Here, the kernel of the host OS allows the existence of multiple isolated user space instances, called containers. The most popular container technology today is Docker[31]. With Docker, the application and its dependencies are packaged in a virtual container that can run on any OS. Docker containers are more lightweight, and a single server or VM can run several containers simultaneously. A docker container image is built by reading instructions from a Dockerfile[32]. This file contains a selection of commands that a user could call on the command line interface to assemble the image. An example of a Dockerfile is shown in Listing 10.10. The image is built from an ubuntu docker image. Then the contents of the `app` directory are copied to the image and the application is built using the make command. Finally the app is started. Containers make it easy to control data and software components, and make frequent updates such as security patches.

Docker promotes the use of multi-container applications, where each service in the application is placed in its own container. This is done through a docker-compose file, as shown in Listing 10.11. This example, used during my research, sets up a mariaDB database and exposes it on port 3306. It also creates a web interface called Adminer, hosted on port 8080.

Placing separate services in their own containers like this improves security, as containers run in isolation by default. Each container can only access ports and files of other containers that are explicitly exposed by them.

Docker increases the level of security in comparison to running applications directly on the host. It has features to more easily encrypt volumes, manage secrets, and encrypt communication between containers, all helping to avoid some of the top categories in the OWASP top 10. But some misconfigurations can still downgrade the level of secu-

[31]`https://www.docker.com/`

[32]`https://docs.docker.com/engine/reference/builder/`

```
version: '3.1'
services:
  maria:
    image: mariadb
    ports:
      - 3306:3306
    volumes:
      - mariadb:/var/lib/mysql
  web:
    image: adminer
    ports:
      - 8080:8080
volumes:
  mariadb:
```

Listing 10.11: Example of a Dockerfile to build and run a Python application.

rity and even introduce new vulnerabilities. Many guides and training exist help developers secure Dockerfiles and other container technology, including the SCW portal, the OWASP website[33], and NIST[34]. Some security tools, like Snyk, have adapted to scan for container misconfigurations, to detect, for example, use of open-source images with known vulnerabilities.

Runtime orchestration

With runtime orchestration, the management of multiple physical servers is being abstracted as well. An orchestration framework exposes a server cluster as if it were a single pool of resources, and allows the installation and management of containers across these servers from one centralized host. Several runtime orchestration frameworks exist, with the most popular being Kubernetes (K8s), originally designed by Google and now maintained by the Cloud Native Computing Foundation (CNCF). Runtime orchestration makes it easy to apply security practices such as network encryption, authentication, and management of application secrets[35].

Kubernetes is designed to be highly customizable and developers must turn on certain features to make sure the resulting configuration

[33] https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
[34] https://csrc.nist.gov/publications/detail/nistir/8176/final
[35] https://kubernetes.io/docs/concepts/security/overview/

```json
{
    "management_chain": {
        "bjorn": [
            "pieter",
            "bert"
        ],
        "alex": [
            "gillis"
        ]
    }
}
```

Listing 10.12: Management chain data example for use in OPA.

is secure. More information can be found on the OWASP website[36].

10.5.2 Policy as code

With policy as code, isolation and decoupling are applied to some of the logic of the applications. A separate service is deployed that can be queried to make policy decisions. One framework to run such a service is Open Policy Agent (OPA), backed up by the CNCF.

To explain how policy decisions can be decoupled, I use the example of authorization. In a budgeting application, a manager may be able to access the salary of anyone who reports to them. To isolate the necessary decisions from the application itself, the management chain can be stored in the policy agent. In Listing 10.12, an example is shown of how such a management chain can be stored in OPA.

It is then possible to define rules and execute queries based on this data. In Listing 10.13, rules are shown that determine who is able to access a salary. Users are allowed to see their own salary and that of other users below them in the management chain.

The budgeting application can then make decisions by querying the OPA service as shown in Listing 10.14.

Decoupling policy decisions has many advantages. It provides a centralized overview of policies and avoids redundancy in implementations. In another application, for example, employees might be able to request absence and these requests can be authorized by their manager, requiring the same management chain to make authorization decisions. Instead of implementing these decisions in the second application and possibly

[36]`https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html`

```
default allow = false

allow {
    input.method = "GET"
    input.path = ["salary", id]
    input.user_id = id
}

allow {
    input.method = "GET"
    input.path = ["salary", id]
    managers = data.management_chain[id]
    input.user_id = managers[_]
}
```

Listing 10.13: OPA rules that define who has access to the salary of other users.

```
> input := {"method": "GET", "path": ["salary", "alex"], "user_id": "
    gillis"}
> allow
false
```

Listing 10.14: Management chain data example for use in OPA.

even storing the data multiple times, the OPA service can be reused by adding new rules.

Because of the declarative nature of the rules, they can be understood easily, causing reduced complexity and hence reduced chance of mistakes. The policy agent can be used to overcome several vulnerabilities in OWASP top 10, such as authentication and authorization flaws, as well as some business logic flaws.

By using a policy as code service, logging of the policy decisions will also be done separately from application logs. This can make it easier to monitor and detect abnormalities, partly mitigating the security problem of excessive logging in the application.

Chapter 11

Conclusion

In this work, the focus is on improving the usability of the developer with more targeted, role-specific training and tools. I propose a methodology to improve collaboration between developers and security professionals, called the paved path methodology. In line with this methodology, I made improvements to both training and tools provided by Secure Code Warrior (SCW).

If nothing else, take away from this chapter...

In the first part of this book, I designed and implemented an Intelligent Tutoring System (ITS). First, the trained two-parameter logistic model (2PL) model from the field of Item Response Theory (IRT) offered some useful insights in the mental model of the developer. To speed up the slow estimation procedure for this model, I created an approximation method that remained sufficiently accurate long after initial calibration. This approximation enables an accurate and easy-to-implement way to adapt existing Collaborative Filtering (CF) algorithms to learning systems.
In part II, I evaluated the Integrated Development Environment (IDE) plugin, called Sensei. The results from the experiments show that customized rules and real-time remediation guidance result in high engagement from the developers using the tool. To encourage this customization of rules, a YAML Ain't Markup Language (YAML) based syntax and User Interface (UI) were designed and evaluated. Security professionals reported that, through this UI, customization of rules is easier than with comparable tools.

11.1 Intelligent Tutoring System

The SCW training platform is usable and relevant to the needs of the developers. It provides defense training, teaching developers to recognize insecure code patterns and how to fix them. The training is available in a wide variety of programming languages and frameworks. However, there is still a significant part of users that only follow a minimal amount of training. It is likely that the pacing of the predetermined courses and tournaments does not fit their needs. This is confirmed through surveys, where a portion of the users indicate they feel bored due to too much repetition or frustrated because the content is moving too fast.

In this work, I aimed to improve the efficiency of the training by adapting the learning pace to each individual user. To do this, I designed an ITS consisting of three algorithmic components. These are exercise selection, user ability estimation, and exercise difficulty estimation.

The latter two are achieved through calibrating a two parameter logistic (2PL) model from the field of IRT. With the results of this model, I have contributed insights into the mental model of a developer. I have shown that both the vulnerability category and the language and framework have a medium-sized effect on the difficulty of an exercise. Some key findings are that languages that require memory management result in increased difficulty, as well as the use of frameworks. For the vulnerability category, results showed that the size of the related code fragments is a big indicator of the difficulty. Design flaws that generally require fixing of larger code fragments, showed a higher difficulty on average.

Because the training of the 2PL model is slow, especially for large data sets, I developed an approximation procedure that outperformed several existing approximation methods from literature. This method could be useful in many situations where full calibration procedures are not appropriate.

Finally, the recommendation system itself was developed and evaluated. In the set of CF algorithms that were tested, model-based algorithms were outperformed by simpler memory-based algorithms. This is likely because the advantages of model-based algorithms such as improved capacity to deal with data sparsity, scalability, and synonyms are not very applicable to the data set of the SCW training platform.

One of the main contributions of this work to the state of the art, is the proposed adaptation of CF algorithms to learning systems. These simple and easy-to-implement adaptations are applicable to many, if not all, CF systems and have shown a significant increase in prediction performance for all algorithms that have been tested.

Whether or not this recommendation system leads to increased engagement remains to be tested after the ITS is implemented into the SCW platform. This process will be gradually completed in several distinct steps. In the future, it is also the intention to extend the ITS so that it uses data from integrations with other developer tools to provide even more targeted training recommendations. Finally, the proposed adaptations to learning systems could be tested on more CF algorithms and more datasets.

The results from the 2PL model, however, also indicate that a gap exists between knowledge and practice. Some of the most notorious vulnerability types in practice, such as injection vulnerabilities and Cross-Site Scripting (XSS), are shown to be relatively easy to find and fix in training. This suggests that education alone is insufficient. It is evidently too hard for a developer to keep track of the security of the code at all times while they are focused on the functionality instead. With the right tools, such as Sensei, this burden can be alleviated and the developer can be timely reminded of the security implications of their work.

11.2 Sensei

There have been great advancements in the field of software security, as explained in Chapter 10. Despite this, vulnerabilities still seem to be present in all types of software. This is because advancements in development methodologies have also greatly sped up development, resulting in fast iterative releases. This increased rate of change has made the job of security professionals more challenging. On top of that, security professionals are understaffed and cannot adequately assist each of the developers to fix their code.

A shift left movement is ongoing to attempt to address this problem. In this movement, security becomes a shared responsibility among everyone involved in the Software Development Life Cycle (SDLC). To help developers secure their own code, they are given security training and handed security tools. However, security training and tools are often designed with security professionals in mind. They are using a reactive approach, and scan for vulnerabilities. This approach requires sufficient code and calling context to be completed, which means that developers often have to go back to the code, potentially long after it was initially developed. This clearly does not integrate well in developer workflows, and it is no surprise that developers dislike and often disable these tools.

Instead, they should be handed role-specific tools, so that they can receive proactive guidance that helps them more effectively and efficiently produce secure code.

That is the intent of Sensei. In the related work, we have seen a distinct lack of such security tools designed with the developer in mind. With this work, we have created and evaluated such a tool. I helped with the design and requirements of this tool that is implemented by the engineering team at SCW. With Sensei, we take a fundamentally different approach, enforcing guidelines regardless of the context, as the code is being written. This will help developers produce secure code from the start.

The contributions of this work include an evaluation of this tool and some of its specific features. In the first experiment, I have evaluated the effect of highly customized rules and remediation guidance in the IDE. I was able to compare the results of this experiment to those of similar experiments with other tools. I found that both the customization of rules as well as the quick-fixes in the IDE itself lead to increased engagement and trust from the developers.

Sensei is not the only tool that offers an easy-to-read rule syntax, or features to more easily develop and test rules. However, usability tests indicate that a UI that helps the rule-writer create rules without requiring intimate knowledge of the rule syntax could be an important feature. We are not currently aware of any other tools that provide such a UI. Interviews with security professionals indicate that features in this UI, such as the live preview, are useful and make development of rules for Sensei easier than for comparable tools. They also indicated that the requirement to create a quick-fix for a rule promotes closer cooperation with colleagues that have a development background.

Based on the results of this work and those of related research, I recommend businesses to consider developer usability more closely and to offer different role-specific tools to developers and security professionals in their teams.

Future research could further expand on the security outcome of a tool like Sensei. Previous work indicates that adhering to secure coding guidelines and best practices leads to more secure code [79, 139]. Other work has also shown that more vulnerabilities are eliminated if feedback cycles are shorter [118]. The contributions of this work indicate that when highly applicable remediation guidance is available, developers actively make use of it. In the future, the combination of these findings could be validated to confirm that real-time guidance and quick-fixes lead to more secure code.

11.3 Paved path methodology

New technology on its own will not turn the tide, but the proposed solutions in this work will make it easier to make the required shift in culture. A shift towards more human-centered and empathy-driven software development processes and workflows.

This work started with the introduction of such a process, called the paved path methodology. In this methodology, the security team should gradually build a paved path for developers to follow. Together with the developers, they should build standards and patterns that guide the rest of the development team to develop secure code from the start. The guidelines should be specific, Application Programming Interface (API)-level instructions, that can be easily understood by developers, and avoid security jargon.

By building a paved path, the security professionals can more easily provide a service to developers, instead of forcing security testing on them. A paved path approach like this can be encouraged with a tool like Sensei. Because the security professional that is creating the rules is now forced to also create a fix. To achieve this, they most likely need help from members of the development team or at the very least colleagues with a development background. Interactions like this result in closer collaboration among the teams and hence more mutual empathy.

A shift like this does not happen overnight, and this work on its own might not radically change software development. But with this method, I advocate to *pave the path* towards secure development, and take a step in the right direction. A step in the journey towards a more human-centered future of software security.

Appendix A

Challenges

A.1 Challenge creation

Challenges are created from a secure software application. To this application vulnerabilities are introduced in separate git branches. By adding a single vulnerability, five different challenges (L1-L5) can be created, as described in Section 2.3. Some extra data is needed in order to generate these exercises:

- Category and subcategory of the vulnerability type.
- A textual description explaining the insecurity.
- A textual description explaining the secure version.
- Line numbers of the code fragment containing the vulnerability.
- Sets of line numbers marking alternative code fragments. These code fragments, usually called *chunks*, are used to generate the possible options in a locate (L2) challenge.
- Three additional branches containing alternative but insecure solutions to the insecurity. These branches are used to generate the options to choose from in a fix (L3) challenge.
- A textual description for each insecure solution explaining why the solution is suboptimal.

The difficulty $D \in \{1, 2, \ldots, 100\}$ of a challenge determines how many insecure chunks c_i (correct answers) and secure chunks c_s (incorrect answers) need to be marked. It also determines over how many files f the markings need to be spread. The number of markings and files is determined as shown in Equation A.1.

Table A.1: The maximum points awarded on completion of a challenge depends on its difficulty.

Difficulty	Points awarded
Easy	100
Medium	200
Hard	300

$$c_i = \left\lceil \frac{D}{50} \right\rceil, \tag{A.1a}$$

$$c_s = \left\lceil \frac{D}{5} \right\rceil, \tag{A.1b}$$

$$f = \left\lceil \frac{D}{15} \right\rceil, \tag{A.1c}$$

This difficulty value D is split up in three tiers, a challenge is considered easy if $D \in [0 - 35[$, medium if $D \in [35 - 70[$ and hard if $D \in [70 - 100]$. For any challenge (L1-L5), D is an indication of the probability of a correct blind guess when this challenge is presented as a locate exercise (L2). This is likely not an accurate indication of the actual difficulty, which is dependent on many more factors, among which the vulnerability type, code quality, and code complexity.

A.2 Scoring

A player on the SCW online learning platform is awarded points for completing a challenge successfully. The amount of points depends on the performance of the player and the difficulty of the challenge.

The maximum points a player can gain for the successful completion of a challenge is determined by its difficulty tier, as shown in Table A.1.

The amount of points actually awarded depends on the performance of the player. The player loses points if they need multiple attempts or uses hints to find the correct answer.

Using hints comes with a penalty that decreases the maximum number of points that can still be gained. This penalty stacks for each hint until all hints are used, and the total penalty reaches 100%, which means that no more points can be scored. This is because, ultimately, in every type of challenge the hints start to remove incorrect options until only the correct option is left. No points should be awarded when

Table A.2: The penalties for each hint depend on the type of challenge. Using all hints always results in a total penalty of 100%.

Hint	Penalty (%)		
	Identify	Locate	Fix
1	0	0	-33
2	-5	-50	-33
3	-35	-50	-34
4	-60	—	—

Table A.3: The points awarded for each number of failed attempts in the three scoring methods.

Failed attempts	Points Awarded (%)		
	Forgiving	Default	Aggressive
0	100	100	100
1	60	60	60
2	30	30	0
3	10	0	—
4	5	—	—
5	0	—	—

only the correct option is left. The penalty for using a hint depends on which type of challenge is faced. The reasoning behind this is that some challenge types have more hints available then others. For an identify exercise, for example, it impossible to provide a hint containing a video that explains the vulnerability type in detail, as this would give away the correct answer. An overview of the penalty for each hint in each type of exercise is shown in Table A.2.

The amount of points that is actually awarded upon completing an exercise depends on the amount of attempts that were needed to find the correct answer. How many of the maximum points are still awarded depends on which scoring method is used, as shown in Table A.3.

Appendix B

Games-Howell post-hoc tests

Table B.1: p-values of the Games-Howell post-hoc tests for the vulnerability categories.

Access control	Business logic flaws	0.0359
Access control	Cryptography	0.0110
Access control	CSRF	0.0010
Access control	Injection	0.0010
Access control	Poor authorization	0.0010
Access control	Session Management	0.0010
Access control	Use of vulnerable components	0.0011
Access control	XXE	0.0021
Authentication	Business logic flaws	0.0010
Authentication	CSRF	0.0010
Authentication	DoS	0.0039
Authentication	Injection	0.0010
Authentication	Poor authorization	0.0010
Business logic flaws	Cryptography	0.0010
Business logic flaws	ICS	0.0010
Business logic flaws	Improper platform usage	0.0010
Business logic flaws	Information exposure	0.0010
Business logic flaws	Injection	0.0010
Business logic flaws	Insufficient logging	0.0010
Business logic flaws	Transport layer protection	0.0010
Business logic flaws	Mass assignment	0.0254
Business logic flaws	Memory errors	0.0055
Business logic flaws	Misconfiguration	0.0010
Business logic flaws	Session Management	0.0010
Business logic flaws	URAF	0.0010
Business logic flaws	Use of vulnerable components	0.0010
Business logic flaws	XXE	0.0010
Broken cryptography	Injection	0.0048
Code tampering	Cryptography	0.0020
Code tampering	improper platform usage	0.0121

Continued on next page

Code tampering	Injection	0.0010
Code tampering	Session Management	0.0010
Code tampering	URAF	0.0347
Code tampering	Use of vulnerable components	0.0010
Code tampering	XXE	0.0010
Cryptography	CSRF	0.0010
Cryptography	DoS	0.0010
Cryptography	Transport layer protection	0.0011
Cryptography	Poor authorization	0.0010
Cryptography	XSS	0.0010
CSRF	ICS	0.0010
CSRF	Improper platform usage	0.0010
CSRF	Information exposure	0.0010
CSRF	Injection	0.0010
CSRF	Insufficient logging	0.0010
CSRF	Transport layer protection	0.0010
CSRF	Transport layer protection	0.0068
CSRF	Mass assignment	0.0117
CSRF	Memory errors	0.0010
CSRF	Misconfiguration	0.0010
CSRF	Session Management	0.0010
CSRF	URAF	0.0010
CSRF	Use of vulnerable components	0.0010
CSRF	XSS	0.0010
CSRF	XXE	0.0010
DoS	ICS	0.0088
DoS	Improper platform usage	0.0010
DoS	Information exposure	0.0023
DoS	Injection	0.0010
DoS	Insufficient logging	0.0010
DoS	Transport layer protection	0.0021
DoS	Memory error	0.0277
DoS	Misconfiguration	0.0010
DoS	Session Management	0.0010
DoS	URAF	0.0010
DoS	Use of vulnerable components	0.0010
DoS	XXE	0.0010
ICS	Injection	0.0010
ICS	Poor authorization	0.0010
Improper platform usage	Poor authorization	0.0010
Improper platform usage	XSS	0.0415
Information exposure	Injection	0.0010
Information exposure	Poor authorization	0.0010
Injection	Insecure authentication	0.0021
Injection	Insufficient logging	0.0388
Injection	Transport layer protection	0.0010
Injection	memory	0.0037
Injection	Misconfiguration	0.0010
Injection	Poor authorization	0.0010
Injection	Reverse engineering	0.0357

Continued on next page

Injection	Unintended data leakage	0.0010
Injection	XSS	0.0010
Insecure authentication	XXE	0.0209
Insufficient logging	Poor authorization	0.0010
Transport layer protection	Poor authorization	0.0010
Transport layer protection	Poor authorization	0.0010
Transport layer protection	Session Management	0.0010
Transport layer protection	Use of vulnerable components	0.0010
Transport layer protection	XXE	0.0010
Mass assignment	Poor authorization	0.0010
Memory error	Poor authorization	0.0010
Misconfiguration	Poor authorization	0.0010
Poor authorization	Session Management	0.0010
Poor authorization	Unintended data leakage	0.0060
Poor authorization	URAF	0.0010
Poor authorization	Use of vulnerable components	0.0010
Poor authorization	XSS	0.0010
Poor authorization	XXE	0.0010
Session Management	Unintended data leakage	0.0207
Session Management	XSS	0.0010
Unintended data leakage	Use of vulnerable components	0.0295
Unintended data leakage	XXE	0.0112
Use of vulnerable components	XSS	0.0010
XSS	XXE	0.0010

Table B.2: *p*-values of the Games-Howell post-hoc tests for the frameworks

Angular One	C	0.0251
Angular One	C#.NET Web Forms	0.0485
Angular One	Cobol	0.0093
Angular One	C++	0.0181
Angular One	Go	0.0047
Angular One	Java Android	0.0335
Angular One	Java EE	0.0273
Angular One	JavaScript Angular.io	0.0252
Angular One	JavaScript React	0.0237
Angular One	NodeJS Express	0.0331
Angular One	Objective C iOS	0.0200
Angular One	Python Django	0.0212
Angular One	Swift	0.0237
C	C#.NET	0.0010
C	Java EE API	0.0010
C	Java Servlets	0.0010
C	Java Spring API	0.0010
C	Java	0.0010
C	Pseudocode	0.0010
C	Python Flask	0.0010

Continued on next page

C	Python	0.0124
C	Terraform	0.0010
C#.NET Core	Java EE API	0.0309
C#.NET Core	Java Servlets	0.0010
C#.NET Core	Java Spring API	0.0094
C#.NET Core	Java	0.0017
C#.NET Core	Pseudocode	0.0452
C#.NET Core	Python Flask	0.0050
C#.NET Core	Terraform	0.0076
C#.NET MVC	Cobol	0.0039
C#.NET MVC	Go	0.0010
C#.NET MVC	Java Servlets	0.0010
C#.NET MVC	Java Spring API	0.0209
C#.NET MVC	Java	0.0033
C#.NET MVC	Python Flask	0.0114
C#.NET MVC	Terraform	0.0170
C#.NET	C#.NET Web Forms	0.0093
C#.NET	Cobol	0.0010
C#.NET	C++	0.0010
C#.NET	Go	0.0010
C#.NET	Java Android	0.0137
C#.NET	Java EE	0.0010
C#.NET	Java Servlets	0.0052
C#.NET	JavaScript Angular.io	0.0010
C#.NET	JavaScript React	0.0010
C#.NET	NodeJS Express	0.0025
C#.NET	Objective C iOS	0.0262
C#.NET	Python Django	0.0010
C#.NET	Swift	0.0010
C#.NET Web API	Java Servlets	0.0015
C#.NET Web Forms	Go	0.0037
C#.NET Web Forms	Java EE API	0.0052
C#.NET Web Forms	Java Servlets	0.0010
C#.NET Web Forms	Java Spring API	0.0019
C#.NET Web Forms	Java	0.0010
C#.NET Web Forms	Pseudocode	0.0010
C#.NET Web Forms	Python Flask	0.0010
C#.NET Web Forms	Terraform	0.0021
AWS CloudFormation	Go	0.0389
Cobol	Java EE API	0.0010
Cobol	Java Servlets	0.0010
Cobol	Java Spring	0.0010
Cobol	Java Spring API	0.0010
Cobol	Java	0.0010
Cobol	PhP Symfony	0.0046
Cobol	PLSQL	0.0019
Cobol	Pseudocode	0.0010
Cobol	Python Flask	0.0010
Cobol	Python	0.0010
Cobol	Terraform	0.0010

Continued on next page

C++	Java EE API	0.0010
C++	Java Servlets	0.0010
C++	Java Spring	0.0073
C++	Java Spring API	0.0010
C++	Java	0.0010
C++	PhP Symfony	0.0265
C++	PLSQL	0.0337
C++	Pseudocode	0.0010
C++	Python Flask	0.0010
C++	Python	0.0010
C++	Terraform	0.0010
docker:vanilla	Java Servlets	0.0043
Go	Java EE API	0.0010
Go	Java Servlets	0.0010
Go	Java Spring	0.0010
Go	Java Spring API	0.0010
Go	Java	0.0010
Go	PhP Symfony	0.0010
Go	PLSQL	0.0010
Go	Pseudocode	0.0010
Go	Python Flask	0.0010
Go	Python	0.0010
Go	Terraform	0.0010
Java Android	Java EE API	0.0035
Java Android	Java Servlets	0.0010
Java Android	Java Spring API	0.0011
Java Android	Java	0.0010
Java Android	Pseudocode	0.0014
Java Android	Python Flask	0.0010
Java Android	Terraform	0.0010
Java EE	Java EE API	0.0010
Java EE	Java Servlets	0.0010
Java EE	Java Spring API	0.0010
Java EE	Java	0.0010
Java EE	Pseudocode	0.0010
Java EE	Python Flask	0.0010
Java EE	Python	0.0055
Java EE	Terraform	0.0010
Java EE API	Java Servlets	0.0203
Java EE API	Java Spring	0.0381
Java EE API	JavaScript Angular.io	0.0010
Java EE API	JavaScript React	0.0010
Java EE API	Kotlin	0.0146
Java EE API	NodeJS Express	0.0015
Java EE API	Objective C iOS	0.0035
Java EE API	Python Django	0.0010
Java EE API	Swift	0.0010
Java JSF	Java Servlets	0.0386
Java Servlets	Java Spring	0.0010
Java Servlets	Java Spring API	0.0468

Continued on next page

Java Servlets	Java	0.0206
Java Servlets	JavaScript Angular.io	0.0010
Java Servlets	JavaScript React	0.0010
Java Servlets	Kotlin	0.0010
Java Servlets	Kubernetes	0.0015
Java Servlets	NodeJS Express	0.0010
Java Servlets	Objective C iOS	0.0010
Java Servlets	PhP Symfony	0.0221
Java Servlets	PLSQL	0.0012
Java Servlets	Pseudocode	0.0046
Java Servlets	Python Django	0.0010
Java Servlets	Python	0.0040
Java Servlets	Ruby on Rails	0.0022
Java Servlets	scala:play	0.0128
Java Servlets	Swift	0.0010
Java Servlets	Terraform	0.0337
Java Spring	Java Spring API	0.0126
Java Spring	Java	0.0010
Java Spring	Pseudocode	0.0087
Java Spring	Python Django	0.0185
Java Spring	Python Flask	0.0068
Java Spring	Terraform	0.0110
Java Spring API	JavaScript Angular.io	0.0010
Java Spring API	JavaScript React	0.0010
Java Spring API	Kotlin	0.0046
Java Spring API	NodeJS Express	0.0010
Java Spring API	Objective C iOS	0.0010
Java Spring API	Python Django	0.0010
Java Spring API	Swift	0.0010
Java	JavaScript Angular.io	0.0010
Java	JavaScript React	0.0010
Java	Kotlin	0.0010
Java	NodeJS Express	0.0010
Java	Objective C iOS	0.0010
Java	Python Django	0.0010
Java	Swift	0.0010
JavaScript Angular.io	Pseudocode	0.0010
JavaScript Angular.io	Python Flask	0.0010
JavaScript Angular.io	Python	0.0074
JavaScript Angular.io	Terraform	0.0010
JavaScript React	Pseudocode	0.0010
JavaScript React	Python Flask	0.0010
JavaScript React	Python	0.0047
JavaScript React	Terraform	0.0010
Kotlin	Pseudocode	0.0097
Kotlin	Python Flask	0.0024
Kotlin	Terraform	0.0039
NodeJS Express	Pseudocode	0.0010
NodeJS Express	Python Flask	0.0010
NodeJS Express	Python	0.0482

Continued on next page

NodeJS Express	Terraform	0.0010
Objective C iOS	Pseudocode	0.0113
Objective C iOS	Python Flask	0.0010
Objective C iOS	Terraform	0.0010
PhP Symfony	Python Django	0.0399
Pseudocode	Python Django	0.0010
Pseudocode	Swift	0.0010
Python Django	Python Flask	0.0010
Python Django	Python	0.0015
Python Django	Terraform	0.0010
Python Flask	Swift	0.0010
Python	Swift	0.0072
Swift	Terraform	0.0010

Appendix C

CF algorithm measurements

Table C.1: The Mean Absolute Error (MAE) (smaller is better), Root Mean Squared Error (RMSE) (smaller is better), and Fraction of Concordant Pairs (FCP) (larger is better) for the default configuration of all available CF algorithms. The baseline algorithm is the best performing benchmark. Among the memory-based algorithms, the k-nearest neighbours (k-NN) baseline is the best performing. For the model-based algorithms SVD++ has the most accurate predictions.

	MAE		RMSE		FCP	
	μ_a	σ_{max}	μ_a	σ_{max}	μ_a	σ_{max}
Benchmark Algorithms						
Normal Predictor	0.7782	0.003	0.9761	0.003	0.4999	0.003
Baseline	0.5004	0.002	0.6169	0.003	0.6049	0.004
Memory-based algorithms						
k-NN basic	0.4786	0.002	0.6087	0.003	0.6206	0.005
k-NN with means	0.4747	0.002	0.6086	0.003	0.6157	0.003
k-NN with z-score	0.4771	0.002	0.6145	0.003	0.6144	0.004
k-NN baseline	0.4680	0.002	0.6009	0.003	0.6200	0.004
Slope One	0.4928	0.002	0.6186	0.003	0.5953	0.004
Model-based algorithms						
Co-Clustering	0.4999	0.007	0.6398	0.005	0.5927	0.007
PMF	0.4783	0.003	0.6123	0.003	0.6118	0.005
NMF	0.4835	0.002	0.6130	0.003	0.6018	0.006
SVD	0.4750	0.003	0.6017	0.003	0.6107	0.004
SVD++	0.4591	0.003	0.5911	0.004	0.6184	0.004

Table C.2: Measurements for item-based versus user-based k-NN algorithms. Item-based configurations perform worse for all algorithms.

		MAE		RMSE		FCP	
		μ_a	σ_{max}	μ_a	σ_{max}	μ_a	σ_{max}
k-NN basic	user-based	0.4786	0.002	0.6087	0.003	0.6206	0.005
k-NN basic	item-based	0.5105	0.002	0.6523	0.003	0.6155	0.004
		+6.7%		+7.2%		-1%	
k-NN with means	user-based	0.4747	0.002	0.6086	0.003	0.6157	0.003
k-NN with means	item-based	0.4816	0.002	0.6139	0.003	0.6157	0.003
		+1.3%		+1.0%		-0.0%	
k-NN with z-score	user-based	0.4771	0.002	0.6145	0.003	0.6151	0.004
k-NN with z-score	item-based	0.4822	0.002	0.6184	0.003	0.6144	0.003
		+1.0%		+0.1%		-0.1%	
k-NN baseline	user-based	0.4680	0.002	0.6009	0.003	0.6200	0.004
k-NN baseline	item-based	0.4807	0.003	0.6130	0.003	0.6165	0.004
		+2.6%		+2.0%		-0.1%	

Table C.3: Measurements for each algorithm and similarity metric combination. For each algorithm the baseline similarity metric is the most accurate.

		MAE		RMSE		FCP	
		μ_a	σ_{max}	μ_a	σ_{max}	μ_a	σ_{max}
k-NN basic	MSD	0.5105	0.002	0.6523	0.003	0.6148	0.003
k-NN basic	Cosine	0.5212	0.003	0.6719	0.004	0.5668	0.004
k-NN basic	Pearson	0.4964	0.003	0.6529	0.004	0.5809	0.006
k-NN basic	Baseline	0.4902	0.003	0.6489	0.004	0.5926	0.004
k-NN with means	MSD	0.4816	0.003	0.6138	0.004	0.6159	0.004
k-NN with means	Cosine	0.4835	0.002	0.6155	0.003	0.6157	0.004
k-NN with means	Pearson	0.4601	0.002	0.5989	0.003	0.6259	0.003
k-NN with means	Baseline	0.4521	0.002	0.5934	0.002	0.6348	0.004
k-NN with z-score	MSD	0.4823	0.002	0.6185	0.003	0.6154	0.004
k-NN with z-score	Cosine	0.4840	0.003	0.6195	0.004	0.6157	0.003
k-NN with z-score	Pearson	0.4593	0.002	0.6005	0.004	0.6262	0.005
k-NN with z-score	Baseline	0.4508	0.002	0.5942	0.003	0.6345	0.003
k-NN baseline	MSD	0.4808	0.003	0.6130	0.003	0.6166	0.005
k-NN baseline	Cosine	0.4828	0.002	0.6147	0.003	0.6164	0.005
k-NN baseline	Pearson	0.4593	0.003	0.5982	0.003	0.6272	0.005
k-NN baseline	Baseline	0.4514	0.002	0.5930	0.003	0.6355	0.004

Table C.4: Measurements for all CF algorithms adapted to learning systems. For each algorithm the improvement is computed in comparison to the best non-learning adapted configuration. All of the algorithms are improved by using the ability filter. The best performing algorithm overall is the k-NN baseline algorithm.

	MAE		RMSE		FCP	
	μ_a	σ_{max}	μ_a	σ_{max}	μ_a	σ_{max}
k-NN basic	0.4232	0.002	0.5621	0.003	0.6447	0.004
	-13.7%		-13.4%		+8.8%	
k-NN with means	0.4261	0.002	0.5668	0.003	0.6434	0.005
	-5.7%		-4.5%		+1.3%	
k-NN with z-score	0.4276	0.002	0.5700	0.003	0.6435	0.003
	-5.1%		-4.0%		+1.4%	
k-NN baseline	0.4206	0.003	0.5601	0.004	0.6437	0.004
	-6.8%		-5.5%		+1.3%	
Slope One	0.4719	0.002	0.6029	0.003	0.6070	0.004
	-4.2%		-2.5%		+2.0%	
Co-clustering	0.4940	0.007	0.6353	0.008	0.6042	0.004
	-1.2%		-5.5%		+1.3%	
PMF	0.4598	0.003	0.6015	0.004	0.6245	0.003
	-3.9%		-1.8%		+2.1%	
NMF	0.4614	0.003	0.5987	0.004	0.6151	0.005
	-4.6%		-2.3%		+2.2%	
SVD	0.4555	0.002	0.5899	0.004	0.6219	0.003
	-4.1%		-2.0%		+1.8%	
SVD++	0.4409	0.003	0.5786	0.003	0.6326	0.005
	-4.0%		-2.1%		+2.3%	

Appendix D

Bad code patterns

Table D.1: Bad code patterns, their transitions, and resulting approved pattern.

Related vulnerabilities	Bad pattern	Transition	Approved pattern
Injection flaws (SQL, NoSQL, OS command, GraphQL, LDAP), log forging, XSS	Polluting trusted data	Extract untrusted data	Separated untrusted data
		Constrain value	Constrained untrusted data
XXE, sensitive data exposure, insecure configuration, insecure deserialization	Missing object configuration	Configure	Approved object configuration
	Insecure object configuration	Correct configuration	
		Remove configuration	Approved implicit object configuration
Insecure deserialization, security misconfiguration, broken access control	Missing annotation	Annotate	Approved annotation
	Disapproved annotation	Correct annotation	
OS command injection, XXE, security misconfiguration, open redirects, insecure data storage	Missing action	Add action	Approved actions
	Calling a disapproved action	Replace with alternative	
		Remove action	—

Table D.2: Quality related bad code patterns, their transitions, and resulting approved pattern.

Bad pattern	**Transition**	**Approved pattern**
Neglecting naming conventions	Rename	Following naming conventions
Neglecting structural conventions	Add method	Following structural conventions
	Add super	
	Add field	
Code duplication	Remove	—

Appendix E

Recipe scopes

Class scope The class scope can enable or disable recipes based on the name and/or package of the class itself or based on the name and package of any classes or interfaces it inherits from.

Method scope The method scope enables recipes based on the name of the method. These scopes are mostly used to enforce guidelines in inherited methods. For example, when creating a servlet in Java Enterprise Edition (EE), it is advised to configure some security headers with the `doGet` and `doPost` methods inherited from the `HttpServlet` class. We do this by enforcing a guideline that states that the `addHeader` method should be called with specific parameters to set the required headers. We then limit the scope of this guideline to only be enabled when the class inherits from `HttpServlet` and the method name is `doGet` or `doPost`. Using the YAML syntax many properties of the method can be used for the scope, such as the number of parameters, the types of parameters, the return type, and any annotations added to the method.

File scope The file scope is used to enable recipes based on project file names. This is mostly used for configuration files. This allows us for example to enforce coding guidelines in the Android manifest file, as its name is always `AndroidManifest.xml`. This scope is not yet migrated to the YAML syntax.

Android context scope The Android context scope was created to raise context awareness in Android projects. In the Android manifest a developer can configure capabilities of components such as activities and broadcast receivers regarding their communication towards the OS. They can listen to any other application, or only to authorized applications, or only to the own application. The Android context scope allows us to enable recipes based on the configuration of the relevant component, so that we can enforce different recipes for different levels of exposure. We can for example allow communication of sensitive information between classes that are configured as private components, but not between other classes. This scope is not yet migrated to the YAML syntax.

Android build property scope The Android build property scope can be used to enable recipes based on the build property of an Android project. Mostly this is used to look at the `minSdkVersion` property, to determine what versions of Android the application will be compiled to. Specific versions of Android have specific vulnerabilities, so recipes need to be disabled based on that build properties. This scope is not yet migrated to the YAML syntax.

Appendix F

Security battlecards

In this appendix, I discuss a number of related tools in more depth and give my opinion on them. Some of these tools are no longer supported, and hence can not be actively endorsed, but they are interesting to discuss nonetheless. These tools are indicated by an orange box. I also discuss one tool that is an in-house tool used by Google, and is hence not publicly available, this tool is marked by a gray box. All remaining tools are still actively updated at the time of writing this thesis and have a significant userbase, they are marked in green.

SonarLint by SonarSource ①

`https://www.sonarlint.org/`

SonarLint is a free IDE plugin that focuses on code quality. As explained in this work, code quality and code security are often related and hence some rules exist in SonarLint that target security rules. SonarLint is developer-friendly as it provides quick-fixes and clear descriptions with small code examples. However, it only provides a small number of security rules and the rules are not easily customized. I definitely recommend SonarLint, but also recommend supplementing it with a more security-focused tool.

Type	Lint	**SDLC**	Develop
Speed	Real-time	**Fix**	Quick-fix

Snyk Open Source

2

`https://snyk.io/product/open-source-security-management/`

Snyk Open Source tests for vulnerabilities in open-source dependencies. It is available in several IDEs but its web view is the most useful. Snyk Open Source provides remediation through automated pull requests to bump the dependency to the latest version. Both GitHub and GitLab have built-in alternatives and I highly recommend using a tool like this.

Type	Software Component Analysis (SCA)	**SDLC**	Build
Speed	Seconds	**Fix**	Pull request

Snyk Code

3

`https://snyk.io/product/snyk-code/`

Snyk Code (formerly DeepCode.ai) claims to be a developer-first static analysis tool. It is a plugin available for JetBrains IDEs and Visual-Studio Code. It works by uploading the code to a cloud service that runs the analysis, which are then displayed in the IDE. While they claim the scans are real-time, in reality I have found that even for very small projects they already take several seconds. The rules cannot be customized as they are generated by machine learning based on open-source commits. Remediation is offered in the form of code examples from these open-source projects on GitHub. These are not always great examples, and sometimes they are even different from the text description. However, the results are usually easy to understand and apply.

Type	SCA	**SDLC**	Develop
Speed	Seconds	**Fix**	Code examples

Dependabot (4)

https://dependabot.com/

Dependabot creates pull requests to keep your dependencies secure and up-to-date. It is acquired by GitHub and is since free to use and integrated into the platform.

Type	SCA	**SDLC**	Build
Speed	Seconds	**Fix**	Pull request

GitLab Dependency Scanning (5)

https://gitlab.com/gitlab-org/security-products/dependency-scanning

GitLab's integrated dependency scanner supports many languages and package managers. It provides remediation through automated merge requests, GitLab's term for pull requests.

Type	SCA	**SDLC**	Build
Speed	Seconds	**Fix**	Merge request

FindBugs (6)

http://findbugs.sourceforge.net/

FindBugs is a static analysis tool that looks for bugs in Java code. It has not been updated since 2017 and its spiritual successor is SpotBugs. Its IDE plugin is not compatible with newer versions of IntelliJ. To customize the rules, APIs must be used. A popular plugin exists, Find Sec Bugs, that is still updated. This plugin customizes the rule set and adds over 100 security bugs.

Type	Static Application Security Testing (SAST)	**SDLC**	Test
Speed	Minutes	**Fix**	Description

SpotBugs

7

`http://findbugs.sourceforge.net/`

SpotBugs is a community supported successor of FindBugs. It is free to use and can find up to 400 bug patterns in Java code. Its IDE plugin is still compatible with the newest version of IntelliJ but the Find Sec Bugs rules can not be easily added to this IDE plugin. SpotBugs is the spiritual successor of FindBugs, carrying on from the point where it left off with support of its community. SpotBugs' bug descriptions are very short, do not suggest any remediation but provide links to relevant Wikipedia articles.

This lack of information and remediation to the developer has shown to result in low developer trust, as was explained in Section 7.2.1. Experiments with this tool showed that half of the reported issues are never even reviewed [116]. SpotBugs is well researched [116, 165, 166] and used in industry. It is also used at the company of one of our trials.

Research by Ayewah et al. [116] showed that the tool has an Effective False Positive (EFP) rate of 77%, and that the most interesting bugs were found and fixed without SpotBugs, namely after they were revealed by static analysis scans later in the SDLC. Ayewah et al. conclude, however, that the tool could have been used to discover those bugs earlier, if only it would have been used more actively by developers. This is in line with our findings indicating that a low EFP rate inhibits effectiveness. I believe that shorter scan times, better descriptions, and remediation help as available in Sensei might improve the use of SpotBugs by developers in earlier stages of development.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Description

Semmle

8

`https://semmle.com/`

Semmle is a tool acquired by GitHub that also offers an IDE plugin. It uses a reactive approach, and the default rules are focused on finding vulnerabilities, but the rules can be customized to use more local analyses and enforce code best practices. Creating new rules is done through a custom query language, called CodeQL. A “Query console” is provided that has syntax highlighting and completion, two features that make development of these queries easier. Some example projects are available in the query console that can be used to test newly developed queries, but in my experience you have to be lucky to find relevant code.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	None

Semgrep

https://semgrep.dev/

9

Out of all the tools reviewed in this work, Semgrep most closely resembles Sensei. Semgrep is a fast static analysis tool commercialised by r2c (`https://r2c.dev/`) that uses pfff (`https://github.com/returntocorp/pfff/`) as a static analysis engine. Semgrep allows easy customization of rules through a YAML syntax that is very similar to that of Sensei. It provides advanced features to tune rules so that they minimize the chance for EFPs. Semgrep rules also include fixes, and use so-called metavariables to reuse parts of the original code, which seems more user-friendly than the moustache code provided by Sensei.
To develop new Semgrep rules, a web-interface called the Playground can be used. In this interface rules can be developed and tested on any fragment of code, which makes development a smooth experience. It also makes it easy to share examples and working rules among the community. A large set of default and community rules are available, most of which target security issues. No real, fleshed-out UI is available to create rules, and the rule-writer is required to read documentation to fully understand the rule syntax. In our research, this has shown to be a hurdle for developers who can be thrown of by learning a new syntax for this purpose. Security professionals, on the other hand, will find that this format is already a big improvement over many other security tools. I found myself able to create moderately complex rules reasonably fast, and I found great community support on the r2c community slack channel.
Despite marketing material using the term "paving the road", Semgrep is still lacking as a developer tool to support the paved path methodology outlined in this work, as it is mostly intended as a Continuous Integration (CI) tool and is hence not a role-specific tool targeted at developers. Third party plugins are available for some IDEs, but they are not officially supported. At the time of writing, the IntelliJ IDEA plugin neither supports the newest version of IntelliJ nor the newest version of Semgrep.

Type	SAST	**SDLC**	Test
Speed	Seconds	**Fix**	Quick-fix

Checkmarx

10

`https://checkmarx.com/`

CxSAST [167] is the static analysis tool by Checkmarx that perfoms source code scans. It has support for over 25 coding and scripting languages, including Java, C#, and python. CxSAST has IDE plugins for Eclipse, Visual Studio, and IntelliJ. In contrast to Sensei, these plugins do not perform any local scans but instead allow uploading the source code to CxSAST. They provide an interactive way to view the scan results by marking the relevant code in the IDE editor. Vulnerabilities marked in the scan results have a category but no descriptions are provided. This means that little help is provided compared to the remediation suggestions and quick-fixes of Sensei. This way developers can learn about the vulnerability and how to fix it in an interactive way. This integration is similar to the integration between Secure Code Warrior and Fortify on Demand or Sensei.
Checkmarx claims flexible rules lead to higher accuracy, and the tool uses a very extensive Query Language (CxQL) [168] to allow the creation of rules. It uses regular Java syntax and is easy to understand. There is good documentation available but no proper tool to create or test rules, this makes development of the rules notably more difficult. It seems that the rules in CxQL can be built to ignore context and enforce secure coding guidelines in line with the paved path methodology. It is also possible to tune CxSAST so that it uses more lightweight analyses, resulting in faster feedback.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Guidance

Tricorder (11)

https://research.google/pubs/pub43322/

Tricorder [107] is a data-driven program analysis platform integrated into the workflow of developers at Google. Tricorder's design philosophy closely resembles that of Sensei where they put developer usability first. Custom analyzers are written in Java, C++, Python, or Go, and also require setting up a service in a docker file.
The results of Tricorder analyzers are shown in a review tool. In this tool quick-fixes are available, but empirical observations have shown they are not used frequently, as only a 20% "Apply fix" rate is reported for Tricorder [107]. It is hypothesised that developers prefer to go back to their IDE to fix the problems [107]. After carefully improving their analyzers, Tricorder reached an EFP rate of around 5%. While both the customized rules of Sensei and the customized analyzers used by Tricorder appear to be effective solutions for preventing EFPs, quick-fixes are more practical in the IDE than during the test or review stage, as is evident from the low "Apply fix" rate for Tricorder.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Quick-fix

Shipshape (12)

https://github.com/google/shipshape

Shipshape is the open-source version of Tricorder (battlecard 11).

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Quick-fix

Fortify Static Code Analyzer (13)

`https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer`

Micro Focus Fortify is an ecosystem that embeds application security testing into all stages of the development tool chain.

As the name suggests, Fortify Static Code Analyzer (FSCA) [169] performs static code analysis on the source code. It can be built in Continuous Integration and Continuous Delivery (CICD) tools and has support for 25 programming languages including Java and C#. Scanning takes several minutes and the results can be shown in a web interface or in integrations with many bug tracking systems, ticketing systems, and code repositories. Fortify recommends using their rule sets that cover over 1000 vulnerability categories and more than one million APIs. Creating new rules can be done in their custom Extensible Markup Language (XML) format in any text editor [170]. Doing so requires reading extensive documentation and learning the proper syntax. They do not provide a rule editor, instead the rule writer can use any preferred text editor.

FSCA provides detailed descriptions of vulnerabilities, which focus on explaining the vulnerabilities in detail, in part by providing examples of insecure code.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Description

Fortify on Demand

14

https://www.microfocus.com/en-us/cyberres/application-security/fortify-on-demand

Fortify on Demand (FOD) [171] provides similar features to FSCA (battlecard 13) but through a web portal, Micro Focus calls this “Application Security Testing as a Service”. It provides the same feedback as FSCA, but in a second tab, also provides a description and code examples to resolve the vulnerability. Both tools also provide links to reference material and to recommended solutions, but on top, FOD provides links to Secure Code Warrior to provide training on a specific vulnerability. An FOD plugin is available for Eclipse, Visual Studio, and IntelliJ. It allows the developer to request static assessments from FOD by uploading the code and downloading the results.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Description

Fortify Security Assistant

15

https://marketplace.microfocus.com/fortify/category/plugins

Fortify Security Assistant (FSA) [172] is a plugin currently available for Eclipse and Visual Studio. It allows security scans similar to that of FSCA (battlecard 13) and FOD (battlecard 14), but does so in the IDE. The rule set is tuned such that the longest analyses are disabled by default. The scan can take several minutes during which the developer cannot make any code changes. This is still quite long compared to the real-time results of Sensei and might inhibit developers from requesting scans frequently during development.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Description

Veracode Static Analysis (16)

`https://www.veracode.com/products/binary-static-analysis-sast`

Veracode Static Analysis (VSA) is a Software as a Service (SaaS) platform that allows the developer to upload their code to be analyzed. The tool performs static analysis scans on compiled bytecode of web applications in 23 programming languages. Because it does not need access to the source code it can also analyse frameworks and libraries used in the project. The downside to this approach is that sufficient code needs to be finished and a successful build is required.
Veracode focuses heavily on detecting vulnerabilities but also guides remediation. To that extent, they provide detailed instructions and videos. There is even the possibility to schedule a one-on-one conference call with a consultation expert.
The company claims most scans finish in under an hour. This means the feedback cycle is rather long compared to the other tools. Since the scans are performed on binaries, they are not able to provide quick-fixes as Sensei does, which is unfortunate for a solution otherwise very focused on remediation.

Type	SAST	**SDLC**	Test
Speed	Hour	**Fix**	Descriptions

Veracode Greenlight (17)

`https://help.veracode.com/r/c_master_greenlight`

Veracode Greenlight (VG) is an IDE plugin that performs lightweight versions of the analyses performed by VSA (battlecard 16). In my opinion, VG is not suited as a developer tool. The lack of rule customization, absence of quick-fixes, and the fact that it analyzes bytecode are all big hurdles for it to be well-integrated in the developer workflow.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Descriptions

Ruleguard

18

`https://go-ruleguard.github.io/`

Ruleguard is an analysis tool for the Go programming language that runs dynamically loaded rules written in Go. Its rules are not restricted to the Abstract Syntax Tree (AST), as it can even match comments. It is not integrated in the IDE but does provide quick-fixes. They can be invoked with the ‘-fix‘ argument when running Ruleguard in the terminal. It has been shown that quick-fixes outside the IDE are not frequently used by developers [107]. The lack of IDE support, and the fact that rule customization is done through a programming language makes this tool less than ideal for use in the paved path methodology.

Type	Linter	**SDLC**	Test
Speed	Seconds	**Fix**	Descriptions

OpenRewrite

19

`https://github.com/openrewrite/rewrite`

OpenRewrite focuses on code refactoring, and the focus is mostly on quality. Its rules always include a “fix” part. Rules are defined in a YAML format, but allow only to match a few basic building block transformations, such as “Change method name”, “Remove annotation”. This closely resembles the original approach taken in the Sensei rule editor, where separate models were created for each such scenario. This quickly caused the amount of models to be unnecessary large and difficult to distinguish from one another. The amount of building blocks supported by OpenRewrite is very limited, however. There is no “Change method argument” model available, which would be required to detect use of the Data Encryption Standard (DES) algorithm as is the running example in this work. The documentation describes how a new model can be developed using their API.

Type	Refactor	**SDLC**	Develop
Speed	Seconds	**Fix**	Quick-fix

OWASP ASIDE

20

https://wiki.owasp.org/index.php/OWASP_ASIDE_Project

The OWASP ASIDE/ESIDE [173] project consist of two branches, the ASIDE branch that focuses on detecting software vulnerabilities and helping developer write secure code, and the ESIDE branch that focuses on helping students in acquiring secure programming knowledge and practices.
ASIDE stands for Application Security IDE (another source claims it is an abbreviation for Assured Software IDE). It performs fast scans of the code in Eclipse, but unlike Sensei the scans need to be started manually. Besides detecting vulnerabilities they also provide quick-fixes for some issues. The quick-fixes require the developers to choose from a list of options, which could overwhelm them. In previous research a large number of false positives were detected [117], however, most of these are what is considered protection for future use in this work. They are cases where best practices should be applied even if their violation is not yet exploitable at this point in development. ASIDE also marks variables in the code that are tainted, this could be compared to Sensei's concept of untrusted input. Untrusted input in Sensei is not currently marked to avoid unnecessary clutter.
The goal of ESIDE [105, 174] is to provide information and training at all times during the education. Its rules can not be configured and the tool does not provide quick-fixes. However, they provide explanations in external web pages linked from Eclipse. Their information is similar to our full coding guidelines where information on APIs and a correct code example is provided.
The project is no longer supported and many of the links on the website are dead.

Type	SAST	**SDLC**	Develop
Speed	Seconds	**Fix**	Quick-fix (sometimes)

SecureAssist

21

https://community.synopsys.com/s/article/SecureAssist-Overview

SecureAssist [161] is an IDE plugin targeting the discovery of security bugs in code. It is available for eclipse, intelliJ, VisualStudio, RAD, and Spring Tool Suite [162]. Its scans are not performed in the IDE but on the enterprise portal. The results are sent back to the IDE once completed. This allows scanning without preventing the developer from continuing his work. Remediation is provided in the form of descriptions that explain the attack and provide some code examples but the tool does not provide quick-fixes [175].
Rule packs are distributed as Java ARchive (JAR) files and the tool provides a Rulepack Configurator similar to Sensei's cookbook Manager. The rules themselves are created in an XML format. No user interface is provided, and writing the rules requires going through long documentation.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	Description

GoKart

22

https://github.com/praetorian-inc/gokart

GoKart is a static analysis tool for Go. It uses specialized techniques for more accurate taint tracking in Go. The analyzers can be easily extended using a YAML format. However, there is no IDE integration and the tool does not offer any form of remediation guidance, so it is not the best suited to give to the developer.

Type	SAST	**SDLC**	Test
Speed	Minutes	**Fix**	None

Bibliography

[1] Trustwave. Global security report. https://www2.trustwave.com/rs/815-RFM-693/images/Trustwave_2018-GSR_20180329_Interactive.pdf. Last accessed 2018-05-22.

[2] U.S. Department of Homeland Security. Infosheet Software Assurance. https://www.us-cert.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf. Last accessed 2018-05-22.

[3] OWASP. Owasp top 10 datacall submissions. https://github.com/OWASP/Top10/tree/master/2017/datacall/submissions. Last accessed 2020-12-19.

[4] Gary McGraw, Sammy Migues, and Jacob West. Building security in maturity model 9(bsimm). https://www.bsimm.com/. Last accessed 2018-05-22.

[5] Gary McGraw, Sammy Migues, and Jacob West. Building security in maturity model 11(bsimm). https://www.bsimm.com/. Last accessed 2020-12-22.

[6] ShiftLeft. Developer productivity & security survey. https://go.shiftleft.io/developer-productivity-and-security-survey. Last accessed 2020-12-22.

[7] StackOverflow. 2020 Developer Survey. https://insights.stackoverflow.com/survey/2020. Last accessed 2021-06-09.

[8] Thomas Lam and Nicolas Chaillan. Department of defense (dod) enterprise devsecops reference design, department of defense (dod) chief information officer. https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf. Last accessed 2021-07-05.

[9] Julie Dirksen. *Design for how people learn.* New Riders, 2015.

[10] Franca Garzotto. Investigating the educational effectiveness of multiplayer online games for children. In *Proceedings of the 6th international conference on Interaction design and children*, pages 29–36, 2007.

[11] Penelope Sweetser and Peta Wyeth. Gameflow: a model for evaluating player enjoyment in games. *Computers in Entertainment (CIE)*, 3(3):3–3, 2005.

[12] Alessandro Febretti and Franca Garzotto. Usability, playability, and long-term engagement in computer games. *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, pages 4063–4068, 04 2009.

[13] Khalid Abed Dahleez, Ayman A El-Saleh, Abrar Mohammed Al Alawi, and Fadi Abdelmuniem Abdelfattah. Higher education student engagement in times of pandemic: the role of e-learning system usability and teacher behavior. *International Journal of Educational Management*, 2021.

[14] Juho Hamari, David J Shernoff, Elizabeth Rowe, Brianno Coller, Jodi Asbell-Clarke, and Teon Edwards. Challenging games help students learn: An empirical study on engagement, flow and immersion in game-based learning. *Computers in human behavior*, 54:170–179, 2016.

[15] Syed Munib HADI and Rebecca RAWSON. Driving learner engagement and completion within moocs: a case for structured learning support. *Proceedings of the European Stakeholder Summit on experiences and best practices in and around MOOCs (EMOOCS 2016)*, page 81, 2016.

[16] Young Ju Joo, Kyu Yon Lim, and Su Mi Kim. A model for predicting learning flow and achievement in corporate e-learning. *Educational Technology & Society*, 15(1):313, 2012.

[17] Mihaly Csikszentmihalyi. Learning,"flow," and happiness. In *Applications of flow in human development and education*, pages 153–172. Springer, 2014.

[18] Giel Van Lankveld, Pieter Spronck, and Matthias Rauterberg. Difficulty scaling through incongruity. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 228–229. AIIDE, 2008.

[19] Janet Maybin, Neil Mercer, and Barry Stierer. Scaffolding learning in the classroom. *Thinking voices: The work of the national oracy project*, pages 186–195, 1992.

[20] Jing Xie, Heather Richter Lipford, and Bill Chu. Why do programmers make security errors? In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 161–164. IEEE, 2011.

[21] Christoph Kern. Securing the tangled web. *Queue*, 12(7):40–55, 2014.

[22] Aniqua Z Baset and Tamara Denning. Ide plugins for detecting input-validation vulnerabilities. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 143–146. IEEE, 2017.

[23] Pieter De Cremer, Nathan Desmet, and Bjorn Madou, Matias De Sutter. Sensei: Enforcing secure coding guidelines in the integrated development environment. *Software: Practice and Experience*, 50(9):1682–1718, 2020.

[24] NIST. Enhancing Software Supply Chain Security. `https://www.nist.gov/itl/executive-order-improving-nations-cybersecurity/enhancing-software-supply-chain-security`. (Archived) Last accessed 2021-11-14.

[25] De Cremer, Pieter and Desmet, Nathan and Madou, Matias and Wong, Colin. Method and system for adaptive security guidance. `https://patents.google.com/patent/US20200211135A1/en`. US20200211135A1.

[26] National Council on Measurement in Education. Glossary (archived). `https://web.archive.org/web/20170722194028/http://www.ncme.org/ncme/NCME/Resource_Center/Glossary/NCME/Resource_Center/Glossary1.aspx?hkey=4bb87415-44dc-4088-9ed9-e8515326a061#anchorC`. Last accessed 2021-09-30.

[27] Mohammad Alshammari, Rachid Anane, and Robert J Hendley. Design and usability evaluation of adaptive e-learning systems based on learner knowledge and learning style. In *IFIP Conference on Human-Computer Interaction*, pages 584–591. Springer, 2015.

[28] Silvia Schiaffino, Patricio Garcia, and Analia Amandi. eteacher: Providing personalized assistance to e-learning students. *Computers & Education*, 51(4):1744–1754, 2008.

[29] Sabine Graf, Silvia Rita Viola, and T Leo Kinshuk. Representative characteristics of felder-silverman learning styles: An empirical model. In *Proceedings of the IADIS International Conference on Cognition and Exploratory Learning in Digital Age (CELDA 2006), Barcelona, Spain*, pages 235–242, 2006.

[30] Richard M Felder, Linda K Silverman, et al. Learning and teaching styles in engineering education. *Engineering education*, 78(7):674–681, 1988.

[31] Man Li, Luosheng Wen, and Feiyu Chen. A novel collaborative filtering recommendation approach based on soft co-clustering. *Physica A: Statistical Mechanics and its Applications*, 561:125140, 2021.

[32] Ritu Sharma, Dinesh Gopalani, and Yogesh Meena. Collaborative filtering-based recommender system: Approaches and research challenges. In *2017 3rd international conference on computational intelligence & communication technology (CICT)*, pages 1–6. IEEE, 2017.

[33] Kai Yu, Anton Schwaighofer, Volker Tresp, Xiaowei Xu, and H-P Kriegel. Probabilistic memory-based collaborative filtering. *IEEE Transactions on Knowledge and Data Engineering*, 16(1):56–69, 2004.

[34] John S Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. *arXiv preprint arXiv:1301.7363*, 2013.

[35] Xiaoyuan Su and Taghi M Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009.

[36] Nicolas Hug. Surprise: A python library for recommender systems. *Journal of Open Source Software*, 5(52):2174, 2020.

[37] Yehuda Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(1):1–24, 2010.

[38] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer, 2011.

[39] Badrul M Sarwar, George Karypis, Joseph Konstan, and John Riedl. Recommender systems for large-scale e-commerce: Scalable neighborhood formation using clustering. In *Proceedings of the fifth international conference on computer and information technology*, volume 1, pages 291–324. Citeseer, 2002.

[40] David Heckerman, David Maxwell Chickering, Christopher Meek, Robert Rounthwaite, and Carl Kadie. Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research*, 1(Oct):49–75, 2000.

[41] María N Moreno, Saddys Segrera, Vivian F López, María Dolores Muñoz, and Ángel Luis Sánchez. Web mining based framework for solving usual problems in recommender systems. a case study for movies recommendation. *Neurocomputing*, 176:72–80, 2016.

[42] Mark O'Connor and Jon Herlocker. Clustering items for collaborative filtering. In *Proceedings of the ACM SIGIR workshop on recommender systems*, volume 128. Citeseer, 1999.

[43] Thomas George and Srujana Merugu. A scalable collaborative filtering framework based on co-clustering. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 4–pp. IEEE, 2005.

[44] Andriy Mnih and Russ R Salakhutdinov. Probabilistic matrix factorization. In *Advances in neural information processing systems*, pages 1257–1264, 2008.

[45] Yu-Xiong Wang and Yu-Jin Zhang. Nonnegative matrix factorization: A comprehensive review. *IEEE Transactions on knowledge and data engineering*, 25(6):1336–1353, 2012.

[46] Patrik O Hoyer. Non-negative matrix factorization with sparseness constraints. *Journal of machine learning research*, 5(9), 2004.

[47] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Application of dimensionality reduction in recommender system-a case study. Technical report, Minnesota Univ Minneapolis Dept of Computer Science, 2000.

[48] Huseyin Polat and Wenliang Du. Svd-based collaborative filtering with privacy. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 791–795, 2005.

[49] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434, 2008.

[50] Chih-Ming Chen, Hahn-Ming Lee, and Ya-Hui Chen. Personalized e-learning system using item response theory. *Computers & Education*, 44(3):237–255, 2005.

[51] Ali O Mahdi, Mohammed I Alhabbash, and Samy S Abu-Naser. An intelligent tutoring system for teaching advanced topics in information security. *World Wide Journal of Multidisciplinary Research and Development*, 2016.

[52] Elizabeth Carter and Glenn D Blank. A tutoring system for debugging: status report. *Journal of Computing Sciences in Colleges*, 28(3):46–52, 2013.

[53] Jihan Y AbuEl-Reesh and Samy S Abu-Naser. An intelligent tutoring system for learning classical cryptography algorithms (ccaits). *International Journal of Academic and Applied Research (IJAAR)*, 2(2), 2018.

[54] Samy S Abu-Naser. Evaluating the effectiveness of the cpp-tutor, an intelligent tutoring system for students learning to program in c++. *Journal of Applied Sciences Research*, 2009.

[55] Antonija Mitrovic. An intelligent sql tutor on the web. *International Journal of Artificial Intelligence in Education*, 13(2-4):173–197, 2003.

[56] Filippos Giannakas, Georgios Kambourakis, and Stefanos Gritzalis. Cyberaware: A mobile game-based app for cybersecurity education and awareness. In *2015 International Conference on Interactive Mobile Communication Technologies and Learning (IMCL)*, pages 54–58. IEEE, 2015.

[57] Ge Jin, Manghui Tu, Tae-Hoon Kim, Justin Heffron, and Jonathan White. Evaluation of game-based learning in cybersecurity education for high school students. *Journal of Education and Learning (EduLearn)*, 12(1):150–158, 2018.

[58] Kristian Beckers and Sebastian Pape. A serious game for eliciting social engineering security requirements. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 16–25. IEEE, 2016.

[59] Affan Yasin, Lin Liu, Tong Li, Jianmin Wang, and Didar Zowghi. Design and preliminary evaluation of a cyber security requirements education game (sreg). *Information and Software Technology*, 95:179–200, 2018.

[60] Michael J Pazzani and Daniel Billsus. Content-based recommendation systems. In *The adaptive web*, pages 325–341. Springer, 2007.

[61] Wentao Kang and Ying Liang. A security ontology with mda for software development. In *2013 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 67–74. IEEE, 2013.

[62] Yan Jia, Yulu Qi, Huaijun Shang, Rong Jiang, and Aiping Li. A practical approach to constructing a knowledge graph for cybersecurity. *Engineering*, 4(1):53–60, 2018.

[63] David J Weiss and G Gage Kingsbury. Application of computerized adaptive testing to educational problems. *Journal of Educational Measurement*, 21(4):361–375, 1984.

[64] David Magis, Duanli Yan, and Alina A Von Davier. *Computerized adaptive and multistage testing with R: Using packages catr and mstr*. Springer, 2017.

[65] Guangming Ling, Yigal Attali, Bridgid Finn, and Elizabeth A Stone. Is a computerized adaptive test more motivating than a fixed-item test? *Applied psychological measurement*, 41(7):495–511, 2017.

[66] George Rasch. Probabilistic models for some intelligence and attainment tests: Danish institute for educational research. *Denmark Paedogiska, Copenhagen*, 1960.

[67] Francesco Bartolucci and Luca Scrucca. Point estimation methods with applications to item response theory models. *International Encyclopedia of Education*, pages 366–373, 12 2010.

[68] Maryam Yarandi, Hamid Jahankhani, Mohammad Dastbaz, and Abdel-Rahman Tawil. Personalised mobile learning system based on item response theory. *Advances in Computing and Technology*, 2011.

[69] Shristi Shakya Khanal, PWC Prasad, Abeer Alsadoon, and Angelika Maag. A systematic review: machine learning based recommendation systems for e-learning. *Education and Information Technologies*, 25(4):2635–2664, 2020.

[70] Barbara G Dodd, RJ De Ayala, and William R Koch. Computerized adaptive testing with polytomous items. *Applied psychological measurement*, 19(1):5–22, 1995.

[71] Arpad E Elo. *The rating of chessplayers, past and present.* Arco Pub., 1978.

[72] Arpad E Elo. Logistic probability as a rating basis. *The Rating of Chessplayers, Past&Present. Bronx NY*, 10453, 2008.

[73] Ben Wise. Elo ratings for large tournaments of software agents in asymmetric games. *arXiv preprint arXiv:2105.00839*, 2021.

[74] Brian Ed Bolton. *Handbook of measurement and evaluation in rehabilitation.* University Park Press, 1976.

[75] Mariola Moeyaert, Kelly Wauters, Piet Desmet, and Wim Van den Noortgate. When easy becomes boring and difficult becomes frustrating: disentangling the effects of item difficulty level and person proficiency on learning and motivation. *Systems*, 4(1):14, 2016.

[76] Alexander Robitzsch, Thomas Kiefer, Margaret Wu, Maintainer Alexander Robitzsch, Wilson Adams, LinkingTo Rcpp, and RcppArmadillo Enhances LSAmitR. Package 'tam'. `https://cran.r-project.org/web/packages/TAM/TAM.pdf`, 2021. Last accessed 2021-10-05.

[77] Hangcheng Liu. *Comparing Welch ANOVA, a Kruskal-Wallis test, and traditional ANOVA in case of heterogeneity of variance.* Virginia Commonwealth University, 2015.

[78] Paul A Games and John F Howell. Pairwise multiple comparison procedures with unequal n's and/or variances: a monte carlo study. *Journal of Educational Statistics*, 1(2):113–125, 1976.

[79] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 109–126, 2020.

[80] Chris Lattner. Chris lattner's homepage. `http://nondot.org/sabre/`. Last accessed 2021-08-18.

[81] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 471–475. SIAM, 2005.

[82] Amos Tanay, Roded Sharan, and Ron Shamir. Biclustering algorithms: A survey. *Handbook of computational molecular biology*, 9(1-20):122–124, 2005.

[83] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[84] Xin Luo, Mengchu Zhou, Yunni Xia, and Qingsheng Zhu. An efficient non-negative matrix-factorization-based approach to collaborative filtering for recommender systems. *IEEE Transactions on Industrial Informatics*, 10(2):1273–1284, 2014.

[85] Cédric Févotte and Jérôme Idier. Algorithms for nonnegative matrix factorization with the β-divergence. *Neural computation*, 23(9):2421–2456, 2011.

[86] Sheng Zhang, Weihong Wang, James Ford, and Fillia Makedon. Learning from incomplete ratings using non-negative matrix factorization. In *Proceedings of the 2006 SIAM international conference on data mining*, pages 549–553. SIAM, 2006.

[87] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295, 2001.

[88] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *International conference on algorithmic applications in management*, pages 337–348. Springer, 2008.

[89] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, NY, USA., 2007.

[90] Netflix. Netflix Prize: Forum / Grand Prize awarded to team BellKor's Pragmatic Chaos. `https://web.archive.org/web/20090924184639/http://www.netflixprize.com/community/viewtopic.php?id=1537`. (Archived) Last accessed 2021-09-23.

[91] Netflix. Netflix Prize: View Leaderboard. `https://web.archive.org/web/20091227111134/http://www.netflixprize.com/leaderboard`. (Archived) Last accessed 2021-09-23.

[92] Yehuda Koren and Joseph Sill. Collaborative filtering on ordinal user feedback. In *Twenty-third international joint conference on artificial intelligence*, 2013.

[93] Dave Wichers and Jeff Williams. Owasp top-10 2017. *OWASP Foundation*, 2017.

[94] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, 2005.

[95] Minzhe Guo and Ju An Wang. An ontology-based approach to model common vulnerabilities and exposures in information security. In *ASEE Southest Section Conference*, 2009.

[96] David E Mann and Steven M Christey. Towards a common enumeration of vulnerabilities. In *2nd Workshop on Research with Security Vulnerability Databases, Purdue University, West Lafayette, Indiana*, 1999.

[97] David W Baker, Steven M Christey, William H Hill, and David E Mann. The development of a common enumeration of vulnerabilities and exposures. In *Recent Advances in Intrusion Detection*, volume 7, page 9, 1999.

[98] Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011 cwe/sans top 25 most dangerous software errors. *Common Weakness Enumeration*, 7515, 2011.

[99] Anuradha Sharma and Praveen Kumar Misra. Aspects of enhancing security in software development life cycle. *Advances in Computational Sciences and Technology*, 10(2):203–210, 2017.

[100] Lars-Ola Damm, Lars Lundberg, and Claes Wohlin. Faults-slip-through—a concept for measuring the efficiency of the test process. *Software Process: Improvement and Practice*, 11(1):47–59, 2006.

[101] Lionel C Briand, Khaled El Emam, Bernd G Freimut, and Oliver Laitenberger. A comprehensive evaluation of capture-recapture models for estimating software defect content. *IEEE Transactions on Software Engineering*, 26(6):518–540, 2000.

[102] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proc. 3rd ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 79–88. ACM, 2008.

[103] Lucas Layman, Laurie Williams, and Robert St Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 176–185. IEEE, 2007.

[104] Matthew Syed. *Black Box Thinking: Why Most People Never Learn from Their Mistakes–But Some Do.* Penguin, 2015.

[105] Michael Whitney, Heather Richter Lipford, Bill Chu, and Tyler Thomas. Embedding secure coding instruction into the ide: Complementing early and intermediate cs courses with eside. *Journal of Educational Computing Research*, 56(3):415–438, 2018.

[106] OWASP Enterprise Security API Toolkits - Datasheet. `https://www.owasp.org/images/8/81/Esapi-datasheet.pdf`.

[107] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 598–608. IEEE Press, 2015.

[108] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[109] Christopher Romeo. How to Transform Developers into Security People. `https://www.rsaconference.com/videos/how-to-transform-developers-into-security-people`.

[110] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proc. 7th workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.

[111] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[112] Lotfi ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, Achim D Brucker, and Philip Miseldine. Factors impacting the effort required to fix security vulnerabilities. In *International Conference on Information Security*, pages 102–119. Springer, 2015.

[113] Maurice Dawson, Darrell Burrell, Emad Rahim, and Stephen Brewster. Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3:49–53, 01 2010.

[114] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[115] Jakob Nielsen and Thomas K Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213, 1993.

[116] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *Companion to the 22nd Conf. on Object-oriented programming systems and applications*, pages 805–806, 2007.

[117] Jing Xie, Bill Chu, Heather Richter Lipford, and John T Melton. ASIDE: IDE support for web application security. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 267–276. ACM, 2011.

[118] Luciano Sampaio and Alessandro Garcia. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software*, 113:337–361, 2016.

[119] C Banerjee and SK Pandey. Software security rules, sdlc perspective. *arXiv preprint arXiv:0911.0494*, 2009.

[120] Madiha Tabassum, Stacey Watson, and Heather Richter Lipford. Comparing educational approaches to secure programming: Tool vs. ta. In *Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[121] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Inferring crypto api rules from code changes. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 450–464. ACM, 2018.

[122] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proc. 31st Int'l Conf. on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[123] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[124] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices*, 51(1):298–312, 2016.

[125] Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman. Bespoke tools: adapted to the concepts developers know. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 878–881, 2015.

[126] Andy Cockburn, Carl Gutwin, Joey Scarr, and Sylvain Malacria. Supporting novice to expert transitions in user interfaces. *ACM Computing Surveys (CSUR)*, 47(2):1–36, 2014.

[127] Marc Sallin, Martin Kropp, Craig Anslow, James W. Quilty, and Andreas Meier. Measuring software delivery performance using the four key metrics of devops. In Peggy Gregory, Casper Lassenius, Xiaofeng Wang, and Philippe Kruchten, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 103–119, Cham, 2021. Springer International Publishing.

[128] Gary McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[129] European Commission. Data protection in the eu. https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en. Last accessed 2021-09-28.

[130] hackerone. What percentage of your software vulnerabilities have gdpr implications? https://www.hackerone.com/sites/default/files/2018-01/GDPR%20Implications-ebook.pdf. Last accessed 2021-09-28.

[131] European Union Agency for Fundamental Rights (FRA). Your rights matter: Data protection and privacy - fundamental rights survey. https://fra.europa.eu/en/publication/2020/fundamental-rights-survey-data-protection. Last accessed 2021-09-28.

[132] Cadelina Cassandra, Yuli Eni, Yuriska Marcela, Stevania Clarissa, et al. Analysis of product trust, product rating and seller trust in e-commerce on purchase intention during the covid-19 pandemic. In *2021 International Conference on Information Management and Technology (ICIMTech)*, volume 1, pages 522–525. IEEE, 2021.

[133] Muhammad Ashraf, Jamil Ahmad, Wareesa Sharif, Arslan Ali Raza, Muhammad Salman Shabbir, Mazhar Abbas, and Ramayah Thurasamy. The role of continuous trust in usage of online product recommendations. *Online Information Review*, 2020.

[134] ISO. Iso/iec 27001 information security management. https://www.iso.org/isoiec-27001-information-security.html. Last accessed 2021-09-29.

[135] National Telecommunications and Information Administration. Software bill of materials. https://www.ntia.gov/sbom. Last accessed 2021-09-28.

[136] Joan Bliss, Mike Askew, and Sheila Macrae. Effective teaching and learning: Scaffolding revisited. *Oxford review of Education*, 22(1):37–61, 1996.

[137] Global Encryption Coalition. Breaking encryption myths. https://www.globalencryption.org/2020/11/breaking-encryption-myths/. Last accessed 2021-09-30.

[138] George Robert Barker, William Lehr, Mark Loney, and Douglas Sicker. The economic impact of laws that weaken encryption. *Available at SSRN 3866902*, 2021.

[139] Heather Richter Lipford, Jing Xie, Will Stranathan, Daniel Oakley, and Bei-Tseng Chu. The impact of a structured application development framework on web application security. In *Proceedings of the 14th Colloquium for Information Systems Security Education*, pages 212–219. Baltimore Marriott Inner Harbor, 2010.

[140] Japan Smartphone Security Association. Android Application Secure Design/Secure Coding Guidebook. `http://www.jssec.org/dl/android_securecoding_en.pdf`. Last accessed 2021-07-25.

[141] Marc Witteman and Martijn Oostdijk. Secure application programming in the presence of side channel attacks. In *RSA conference*, volume 2008, 2008.

[142] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating security and systems engineering.* John Wiley & Sons, 2013.

[143] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot's code contributions. *CoRR*, abs/2108.09293, 2021.

[144] Martin Fowler, Jim Highsmith, et al. The agile manifesto. *Software development*, 9(8):28–35, 2001.

[145] Ken Schwaber. *Agile project management with Scrum.* Microsoft press, 2004.

[146] Floris MA Erich, Chintan Amrit, and Maya Daneva. A qualitative study of devops usage in practice. *Journal of software: Evolution and Process*, 29(6):e1885, 2017.

[147] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[148] Snyk. The state of open source security report 2020. `https://go.snyk.io/SoOSS-Report-2020.html`. Last accessed 2021-09-28.

[149] Daniela Soares Cruzes, Michael Felderer, Tosin Daniel Oyetoyan, Matthias Gander, and Irdin Pekaric. How is security testing done in agile teams? a cross-case analysis of four software teams. In *Int'l Conf. on Agile Software Development*, pages 201–216. Springer, 2017.

[150] Daniela Soares Cruzes, Michael Felderer, Tosin Daniel Oyetoyan, Matthias Gander, and Irdin Pekaric. How is security testing done in agile teams? a cross-case analysis of four software teams. In Hubert Baumeister, Horst Lichter, and Matthias Riebisch, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 201–216, Cham, 2017. Springer International Publishing.

[151] Sven Türpe, Laura Kocksch, and Andreas Poller. Penetration tests a turning point in security practices? organizational challenges and implications in a software development team. In *WSIW@ SOUPS*, 2016.

[152] Lynn Futcher and Rossouw von Solms. Guidelines for secure software development. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, pages 56–65. ACM, 2008.

[153] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[154] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[155] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[156] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[157] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. *ACM SIGPLAN Notices*, 51(4):143–157, 2016.

[158] R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: A controlled experiment. In *24th Int'l Symp. on Software Reliability Engineering (ISSRE)*, pages 451–460, Nov 2013.

[159] SpotBugs. SpotBugs API Documentation. https://javadoc.io/doc/com.github.spotbugs/spotbugs/3.1.10. Last accessed 2019-10-15.

[160] FindSecBugs. Find Security Bugs: The SpotBugs plugin for security audits of Java web applications. https://find-sec-bugs.github.io/. Last accessed 2019-10-15.

[161] Synopsys. SecureAssist Overview. https://community.synopsys.com/s/article/SecureAssist-Overview. Last accessed 2019-10-25.

[162] Synopsys. SAST in IDE (SecureAssist). https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/secureassist-datasheet.pdf. Last accessed 2019-10-25.

[163] Synopsys. SecureAssist Custom Rule Tutorial. http://download.asteriskresearch.com/2.4/SecureAssist%20Custom%20Rule%20Tutorial%2010-2014.pdf. Last accessed 2019-10-25.

[164] Michael Wittig and Andreas Wittig. *Amazon web services in action.* Simon and Schuster, 2018.

[165] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proc. 19th Int'l Symp. on Software testing and analysis*, pages 241–252, 2010.

[166] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sept 2008.

[167] Checkmarx. Static Application Security Testing: Secure Your Code from the Very Beginning. https://www.checkmarx.com/products/static-application-security-testing/. Last accessed 2019-10-15.

[168] Checkmarx. CxAudit Overview. https://checkmarx.atlassian.net/wiki/spaces/KC/pages/5406733/CxAudit+Overview. Last accessed 2019-10-15.

[169] Micro Focus. Fortify Static Code Analyzer: Static Application Security Testing. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview. Last accessed 2019-10-14.

[170] Micro Focus. Fortify SCA Custom Rules Reference. http://bigsec.net/b52/Fortify/rules-schema/. Last accessed 2019-10-15.

[171] Micro Focus. Fortify on Demand: Application Security as a Service. https://www.microfocus.com/en-us/products/application-security-testing/overview. Last accessed 2019-10-15.

[172] Micro Focus. Secure SDLC - IDEs. https://www.microfocus.com/en-us/marketing/secure-sdlc-and-devops#section3. Last accessed 2019-10-15.

[173] OWASP. ASIDE Project. https://www.owasp.org/index.php/OWASP_ASIDE_Project. Last accessed 2019-10-16.

[174] UNC Charlotte College of Computing and Informatics. Educational Security in the IDE (ESIDE). https://eside.uncc.edu/. Last accessed 2019-10-16.

[175] Synopsys. How to use SecureAssist IntelliJ Plugin. https://community.synopsys.com/s/article/How-to-Use-SecureAssist-IntelliJ-Plug-in. Last accessed 2019-10-25.

A silhouette of the author standing behind a binary representation of the title of this book.