BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

INTEGRATION TESTS OF THE FITCRACK SYSTEM IN SELENIUM

INTEGRAČNÍ TESTY SYSTÉMU FITCRACK V PROSTŘEDÍ SELENIUM

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **VIKTOR RUCKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. RADEK HRANICKÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

147223

Institut:	Department of Information Systems (UIFS)
Student:	**Rucký Viktor**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Integration Tests of the Fitcrack System in Selenium**
Category:	Web applications
Academic year:	2022/23

Assignment:

1. Study the principles of application testing and CI/CD issues. Focus on testing graphical user interfaces of web applications in different browsers and the Selenium project.
2. Learn about the architecture and implementation of Fitcrack. Focus on the Fitcrack WebAdmin application.
3. After consultation with your supervisor and consultant, design an abstraction layer to describe the semantics of operations in the WebAdmin application. Using this layer, design a solution for integration testing of the Fitcrack system. In particular, focus on creating jobs of different types, mapping computation nodes, monitoring the progress of the computation, and analyzing the results. The tests should also cover components for managing dictionaries, masks, and other resources.
4. Implement the proposed solution.
5. Experimentally demonstrate the applicability of your tests in a real deployment of the system.
6. Evaluate the results and suggest possible future extensions.

Literature:

- Holmes, Antawan, and Marc Kellogg. "Automating functional tests using selenium." *AGILE 2006 (AGILE'06)*. IEEE, 2006.
- Bruns, Andreas, Andreas Kornstadt, and Dennis Wichmann. "Web application tests with selenium." *IEEE software* 26.5 (2009): 88-91.
- HRANICKÝ Radek. Digital Forensics: The Acceleration of Password Cracking. *Ph.D. thesis*. Faculty of Information Technology, Brno University of Technology. 2022. https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=237112
- HRANICKÝ Radek, ZOBAL Lukáš, VEČEŘA Vojtěch, MÚČKA Matúš, HORÁK Adam, BOLVANSKÝ Dávid a ŽENČÁK
  Tomáš. The architecture of Fitcrack distributed password cracking system, version 2. FIT-TR-2020-04, Brno: Fakulta
  informačních technologií VUT v Brně, 2020.

Requirements for the semestral defence:
Points 1 to 3

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Hranický Radek, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	26.10.2022

Abstract

Fitcrack is a distributed password-cracking system developed at FIT BUT. Fitcrack is chiefly interacted with through a web-based front-end. It is a fairly large project that lacks a set of automatised integration tests. Selenium is a browser-automation project that allows controlling browsers programmatically. This project aims to design a test suite of integration tests for Fitcrack to be implemented using Selenium. The design of the test suite utilised the page-object model, a way to split isolate UI-handling code from tests. The tests are implemented in Python using the Pytest framework.

Abstrakt

Fitcrack je systém pro distribuované lámání hesel vyvíjený na VUT FIT. Fitcrack je používán hlavně skrz webové rozhraní. Fitcrack je relativně velký projekt, kterému ale chybí sada automatických integračních testů. Selenium je projekt pro automatizované ovládání webových prohlížečů. Tato bakalářská práce má za cíl navrhnout sadu integračních testů, které budou implementované pomocí projektu Selenium. Návrh sady testů je založen na modelu page-object, způsob jak oddělit kód pro interakci s uživatelským rozhraním od testů. Testy jsou implementovány v jazyku Python pomocí frameworku Pytest.

Keywords

testing, web testing, page-object model, pytest, Selenium, Fitcrack, password cracking

Klíčová slova

testování, testování webu, model page-object, pytest, Selenium, Fitcrack, lámání hesel

Reference

RUCKÝ, Viktor. *Integration Tests of the Fitcrack System in Selenium.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Hranický, Ph.D.

Integration Tests of the Fitcrack System in Selenium

Declaration

I hereby declare that this term project was prepared as an original work by the author under the supervision of Ing. Radek Hranický, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Viktor Rucký
May 10, 2023

Acknowledgements

Firstly, I would like to thank my supervisor, Radek Hranický. Though the thanking of supervisors is oft thought of among students as a necessity, my thanks here are quite earnest and can only recommend him as thesis advisor. Then I want to thank my family and the students of FIT BUT (especially a specific clandestine group thereof; you know who you are) for creating a comfortable home and university atmosphere respectively, keeping me sane enough while working on this thesis. I'd like to dedicate a part of this acknowledgement section in memory of a dear friend, who sadly passed during the creation of this thesis; Vanir, you are missed.

Contents

Chapter 1

Introduction

Nowadays, software testing is a pivotal part of software development. As codebases grow in size, a single changed line of code may have knock-on effects in other parts of code and cause unforeseen issues. We can mitigate this by having an extensive set of tests available: after every code change, the developer can run the set of tests to see whether their change introduced unwanted and unintended bugs.

Fitcrack is a distributed password-cracking system developed at FIT BUT. It is a software project of considerable scope; however, the project currently lacks extensive integration tests. The goal of this thesis is thus to create a set of integration tests that will test the entire system as a whole. Fitcrack uses a web front-end interface called Webadmin, which is where Selenium comes in. Selenium is a popular software project used for automating web browsers, which is why it is chosen as the platform for writing Fitcrack integration tests.

Testing web applications is a not a trivial matter. The tests must directly interact with the UI of the application, which can be a problem as many tests perform the same actions in the UI, which could lead to large code duplication and difficulty of maintaining the tests. An approach called the page-object model aims to solve this problem. The basis of the approach is splitting the code that interacts with the UI into page objects. Page objects represent a single web page or part of the UI and provide methods that perform actions in the UI. Tests then interact with the application under test exclusively through the page objects representing various parts of the application.

In Chapter 2, the workings of the Fitcrack password-cracking system are explained. Then in Chapter 3, principles of testing web applications are explored, including the page-object model. More concretely thereafter, Chapter 4 deals with how the Selenium browser-automation project works. Chapter 5 explains the design and implementation of the set of page objects representing the Webadmin interface for controlling Fitcrack. In Chapter 6, the design and implementation of tests is described. Lastly, in Chapter 7, the usability of the tests in real situations is evaluated.

Chapter 2

Fitcrack Password-Cracking System

Fitcrack is a distributed cracking system developed within the NES@FIT research group at FIT BUT. It allows for the spreading of password-cracking tasks across multiple computers and managing these tasks from a central point [3, 4].

Password cracking works as follows: One provides a list of hashed passwords to a password-cracking system. The system then tries to discover the passwords that generated those hashes by iterating a set of candidate passwords and generating hashes from them. The set of candidate passwords is commonly called the *keyspace*. If a generated hash matches a hash from the input password-hash list, we have "cracked" the password—we know which password generated the input hash. The keyspace is determined by the user and can be dynamically generated. It is also possible that the keyspace does not contain the password of a hash in the input list—in that case, the hash remains "uncracked", and the user needs to start the process again with a different keyspace. The way how a keyspace is generated is called an *attack mode*—the different attack modes supported by Fitcrack are explained in Section 2.2.

Fitcrack comprises chiefly of two parts: the Fitcrack server and Fitcrack hosts that connect to the server. The Fitcrack server is responsible for the management of cracking jobs; however, the server performs no password cracking by itself. To that end, host machines connect to the Fitcrack server, whose sole job is to perform cracking jobs assigned to them by the Fitcrack server. On the host machines, a tool called hashcat[1] is run to perform the cracking tasks. Fitcrack does not actually contain any password-cracking logic or code by itself—it delegates that work to hashcat—rather, it handles the distribution, management, and creation of cracking tasks across multiple machines; Fitcrack also handles ancillary tasks such as keeping a repository of resource files needed for password cracking.

The Fitcrack server exposes and can be controlled through a REST API. Provided is also Webadmin, a web-based graphical front end, and it is the chief way of interacting with Fitcrack. This chapter explains Fitcrack from the perspective of an end-user using Webadmin.

[1]`https://hashcat.net/hashcat/`

2.1 Using Fitcrack to Perform an Attack

In this section, we will step through using Fitcrack to perform a password-cracking job as a user using the Webadmin front-end.

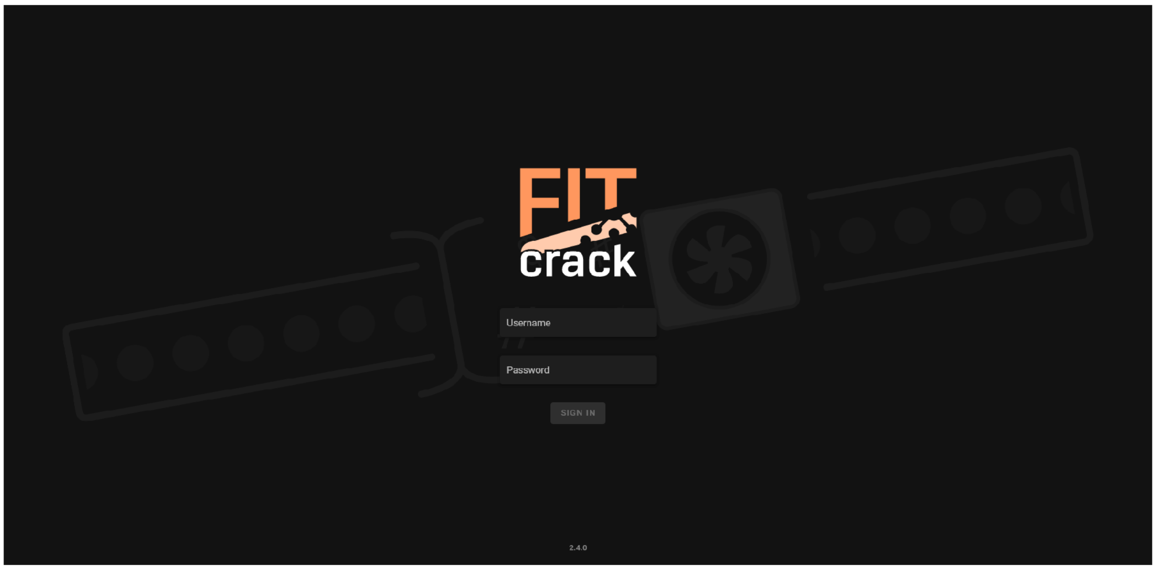
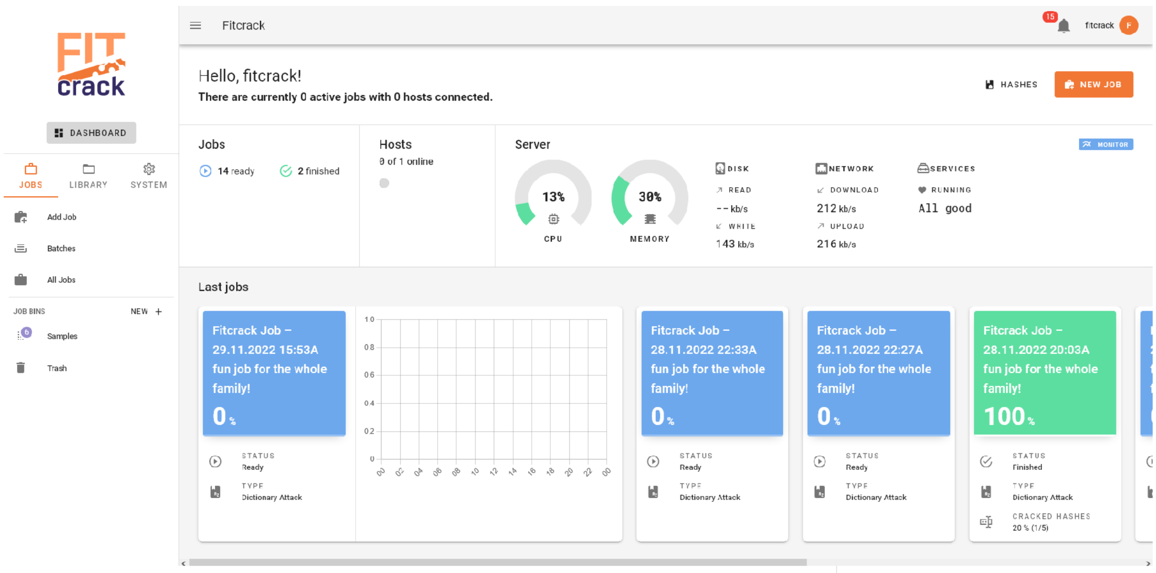


Figure 2.1: Login page of Fitcrack

When we navigate to the homepage of a Fitcrack Webadmin installation, we are greeted with a login screen (Fig. 2.1). The user types their name and password and (if entering correct information), gets navigated to the dashboard (Fig. 2.2).



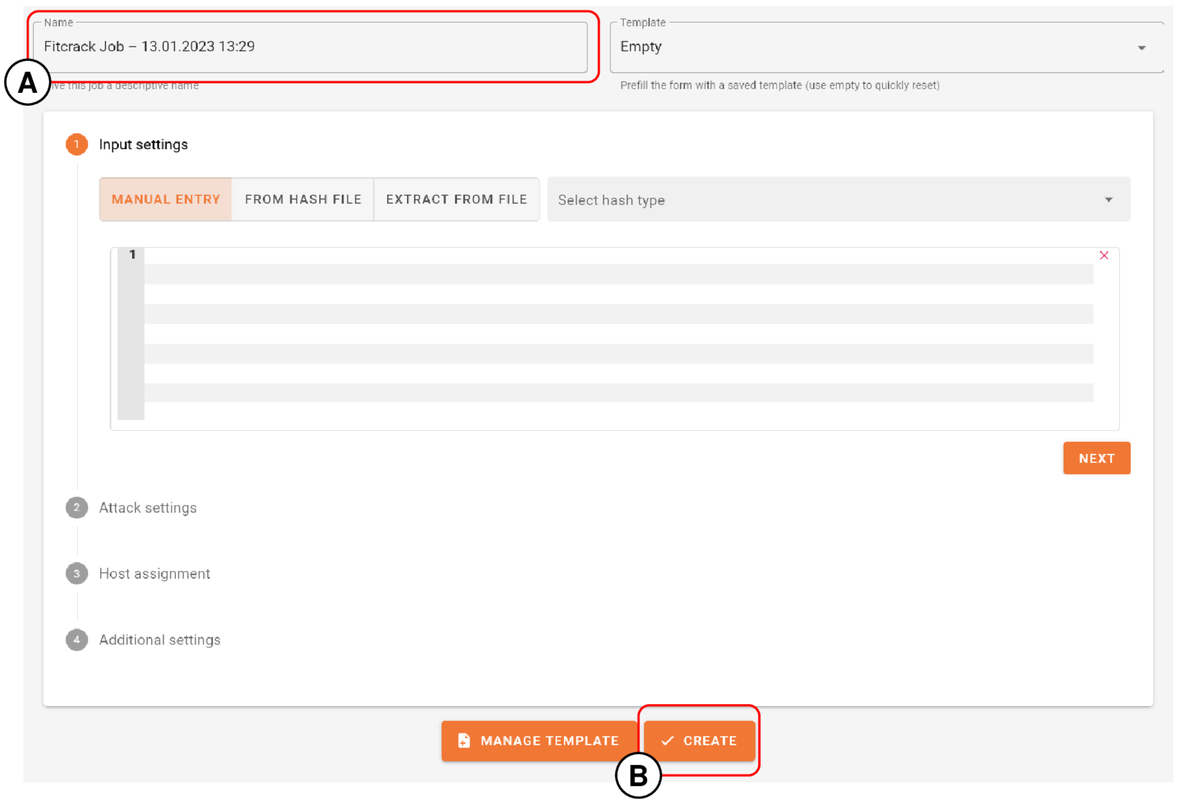
Hello, fitcrack!
There are currently 0 active jobs with 0 hosts connected.

Jobs: 14 ready, 2 finished
Hosts: 0 of 1 online
Server: 13% CPU, 30% MEMORY
DISK: READ -- kb/s, WRITE 143 kb/s
NETWORK: DOWNLOAD 212 kb/s, UPLOAD 216 kb/s
SERVICES: RUNNING All good

Fitcrack Job – 29.11.2022 15:53A fun job for the whole family! 0 % — STATUS Ready, TYPE Dictionary Attack
Fitcrack Job – 28.11.2022 22:33A fun job for the whole family! 0 % — STATUS Ready, TYPE Dictionary Attack
Fitcrack Job – 28.11.2022 22:27A fun job for the whole family! 0 % — STATUS Ready, TYPE Dictionary Attack
Fitcrack Job – 28.11.2022 20:03A fun job for the whole family! 100 % — STATUS Finished, TYPE Dictionary Attack, CRACKED HASHES 20 % (1/5)

Figure 2.2: Dasboard page of Fitcrack, showing an overview of the Fitcrack server

On the dashboard, one can see an overview of all relevant server information, such as the state of the cracking jobs on the Fitcrack system, the status of host computers connected to the Fitcrack server, and the resource (CPU, memory, etc.) utilisation of the machine the


Name: Fitcrack Job – 13.01.2023 13:29

ve this job a descriptive name

Template: Empty

Prefill the form with a saved template (use empty to quickly reset)

Figure 2.3: The job-creation interface of Fitcrack — input for setting the name of the cracking job (A), button that creates the configured cracking job (B)

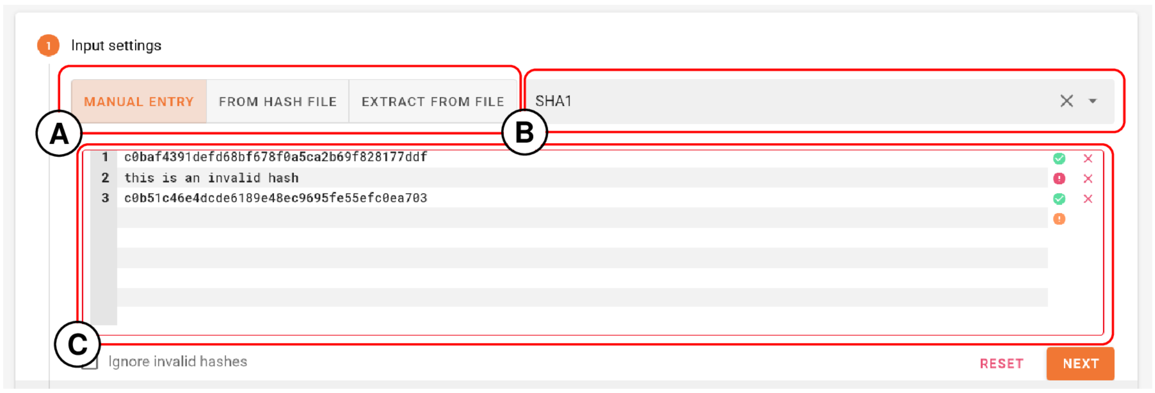
system is running on. To create a cracking job, the user can press the "Add Job" button located on the left in the sidebar, or they can press the "NEW JOB" button in the top right corner of the dashboard.

After being navigated to the job-creation page (Fig. 2.3), one can type in the name of the cracking job into a text box (A). This name is purely for informative and organisational purposes. To configure a cracking job, one needs to go through four stages of job configuration.

First up when configuring a cracking job are the input settings (Fig. 2.4). Here one can set up which hashes one wants to crack (i.e. wants to discover the plaintext passwords that generated them). There are three input modes, which can be selected by pressing their respective buttons (A).

In the manual-entry mode, the user types in hashes one by one into the hash-input text box (C). Fitcrack cannot automatically infer the type of hash entered, so the user has to select the appropriate hash type from a dropdown selection (B). After the selection is made, however, Fitcrack can verify whether the input hashes are valid. The first line in the hash input textbox (B) has a green dot beside it, signifying that it contains a valid SHA1 hash, the second line has a red dot beside it, signifying that it does not contain a valid hash, and lastly the yellow dot beside the third line[2] indicates that the hash has already been cracked in the past and that Fitcrack has its password stored in cache.

[2]The yellow dot is actually beside the fourth line, not the third, but this is errant behaviour captured by the figure. Please imagine that the yellow dot is beside the third line.



```
1  c0baf4391defd68bf678f0a5ca2b69f828177ddf
2  this is an invalid hash
3  c0b51c46e4dcde6189e48ec9695fe55efc0ea703
```

Figure 2.4: The input settings for configuring a cracking job in Fitcrack — input mode selection (A), hash mode selection (B), text box for entering hashes (C)

The second input mode, "from hash file", is quite similar to the manual-entry mode, with the difference that the hashes are not input manually, but are imported from a hash file, where each line contains a hash to be imported. This is pretty much equivalent to copying and pasting the hash file into the hash-input text box in manual-entry mode. As with the manual-entry mode, no hash-type inference is performed, and the hash type has to be selected manually.

The third and final file-extraction mode works by extracting hashes from protected files, like password-protected PDF, ZIP or Microsoft Word files. In this case, the hash type is inferred automatically.

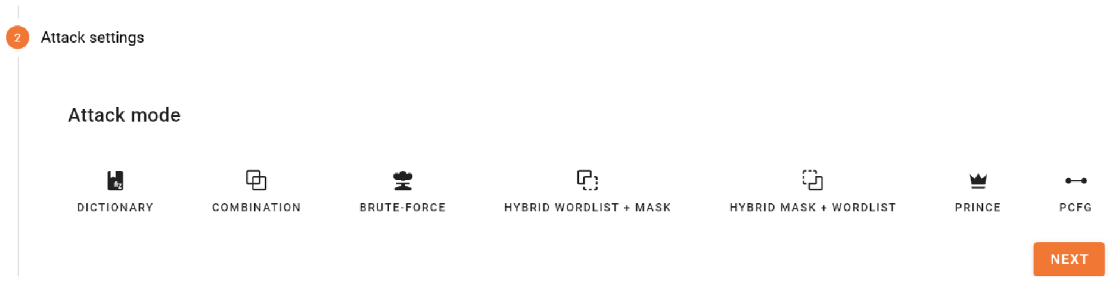
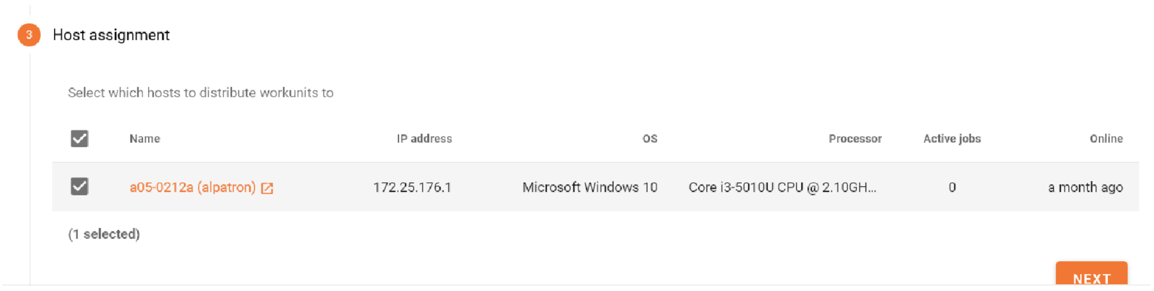


Figure 2.5: The attack-mode selection interface on the job-creation page in Fitcrack

After the hashes are input, the attack-mode configuration is next. In Fig. 2.5, one can select which attack mode is to be used. Configuring attack modes is quite complex and involved and thus is discussed in full in Section 2.2; however, for now, suffice it to say that the attack mode is the way in which the passwords that are checked against the input hashes are generated.

The next-to-last step is assigning hosts to the job. The host-assignment interface (Fig. 2.6) shows a list of host computers connected to the Fitcrack server, and the user must choose at least one computer which will receive the cracking job.

Finally, we can set various options in the last configuration section called "Additional settings" (Fig. 2.7). Here we can set a comment for the cracking job—completely for informational purposes only. We can also set at which time the job should start, and until how late it should at longest run. Interesting is the option for setting the "desired time per workunit": since Fitcrack is a distributed system, it needs to be able to split cracking tasks



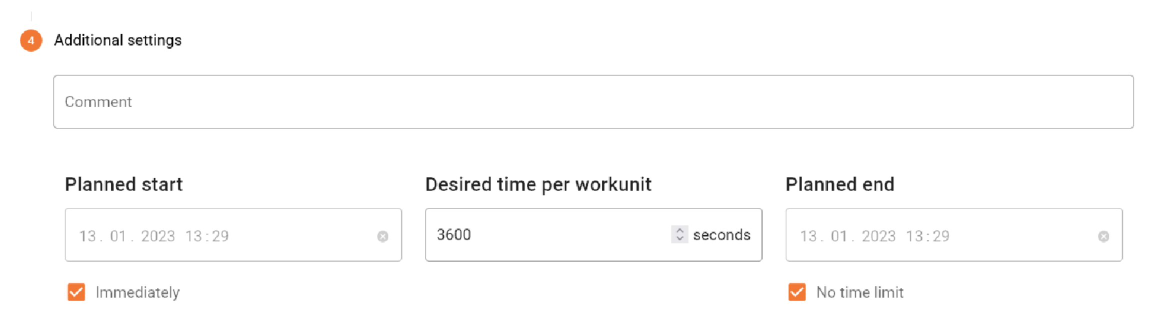
3 Host assignment

Select which hosts to distribute workunits to

✓	Name	IP address	OS	Processor	Active jobs	Online
✓	a05-0212a (alpatron)	172.25.176.1	Microsoft Windows 10	Core i3-5010U CPU @ 2.10GH...	0	a month ago

(1 selected)

Figure 2.6: The host-assignment interface on the job-creation page in Fitcrack



4 Additional settings

Comment

Planned start	Desired time per workunit	Planned end
13. 01. 2023 13:29	3600 seconds	13. 01. 2023 13:29
✓ Immediately		✓ No time limit

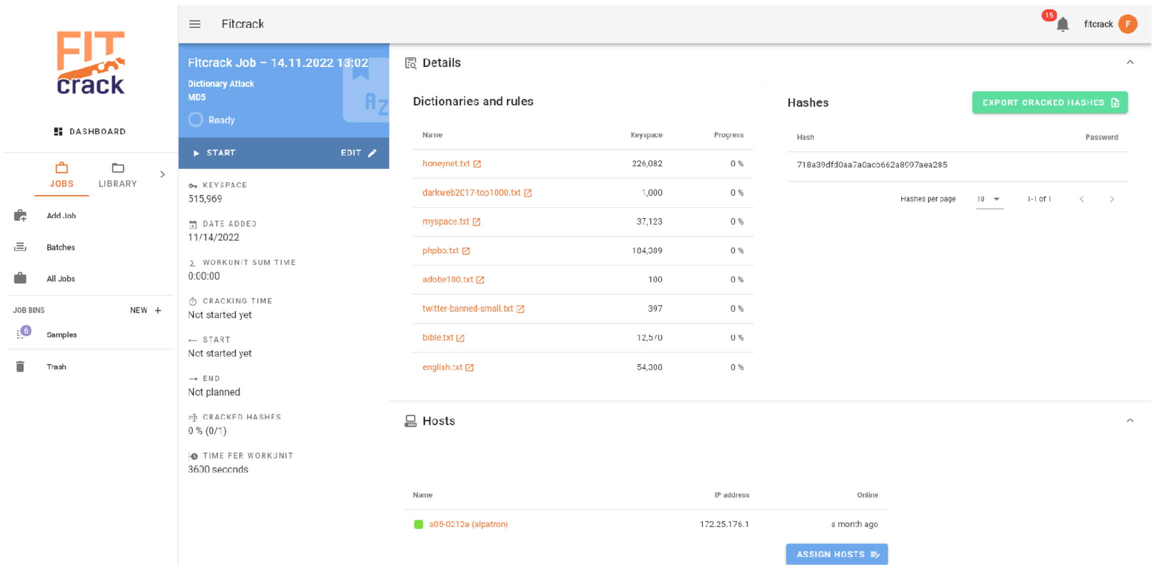
Figure 2.7: The interface for inputting additional settings on the job-creation page in Fitcrack

across multiple computers. A unit of cracking work that is distributed by Fitcrack is called a workunit. Workunits are tailored to every computer that receives them. For example, suppose we want to send a cracking job to two computers—one can check 1,000 passwords per second and the second can check 2,000 passwords per second. And suppose we want to use workunits that take one minute to complete. Given that, Fitcrack will send workunits that contain 60,000 passwords to check to the first computer and workunits that contain 120,000 passwords to the second computer. This way, adverse effects like waiting for a slow computer to finish while all others completed all their tasks can be avoided.[3]

Finally, we can create the configured cracking job by pressing the "Create" button (Fig. 2.3.B). We are then navigated to the job-detail screen of our newly created job (Fig. 2.8). From here, we can start and stop the job, monitor its progress, and in the end see if any of hashes we input have been cracked.

A cracking job is always in a specific state. The states a cracking job can be in are "Ready" (job can be started), "Finished" (job has ended and cracked at least one hash), "Exhausted" (job has ended and did not crack any hash), "Malformed" (job created with malformed input and cannot be run), "Timeout" (job ended early because a timeout was set and it expired), "Running" (job is currently running), and "Finishing" (job is currently running, but all workunits have been distributed and the Fitcrack server is only waiting for results from hosts).

[3]There are more peculiarities concerning workunit distribution, but they are not be discussed in this thesis; see Hranický's dissertation thesis [3, p. 53] for more information about workunit distribution.



Fitcrack Job – 14.11.2022 13:02
Dictionary Attack
MD5
Ready

KEYSPACE
515,969

DATE ADDED
11/14/2022

WORKUNIT SUM TIME
0:00:00

CRACKING TIME
Not started yet

START
Not started yet

END
Not planned

CRACKED HASHES
0 % (0/1)

TIME PER WORKUNIT
3600 seconds

Details

Dictionaries and rules

Name	Keyspace	Progress
honeynet.txt	226,082	0 %
darkweb2017-top1000.txt	1,000	0 %
myspace.txt	37,123	0 %
phpbb.txt	184,389	0 %
adobe100.txt	100	0 %
twitter-banned-small.txt	397	0 %
bible.txt	12,570	0 %
english.txt	54,308	0 %

Hashes

Hash	Password
718a39dfd0aa7a0acb662a8997aea285	

Hashes per page 10 1-1 of 1

Hosts

Name	IP address	Online
🟩 a05-0212a (alpatron)	172.25.176.1	a month ago

Figure 2.8: The job-detail view of Fitcrack showing the details of a created cracking job

2.2 Attack Modes

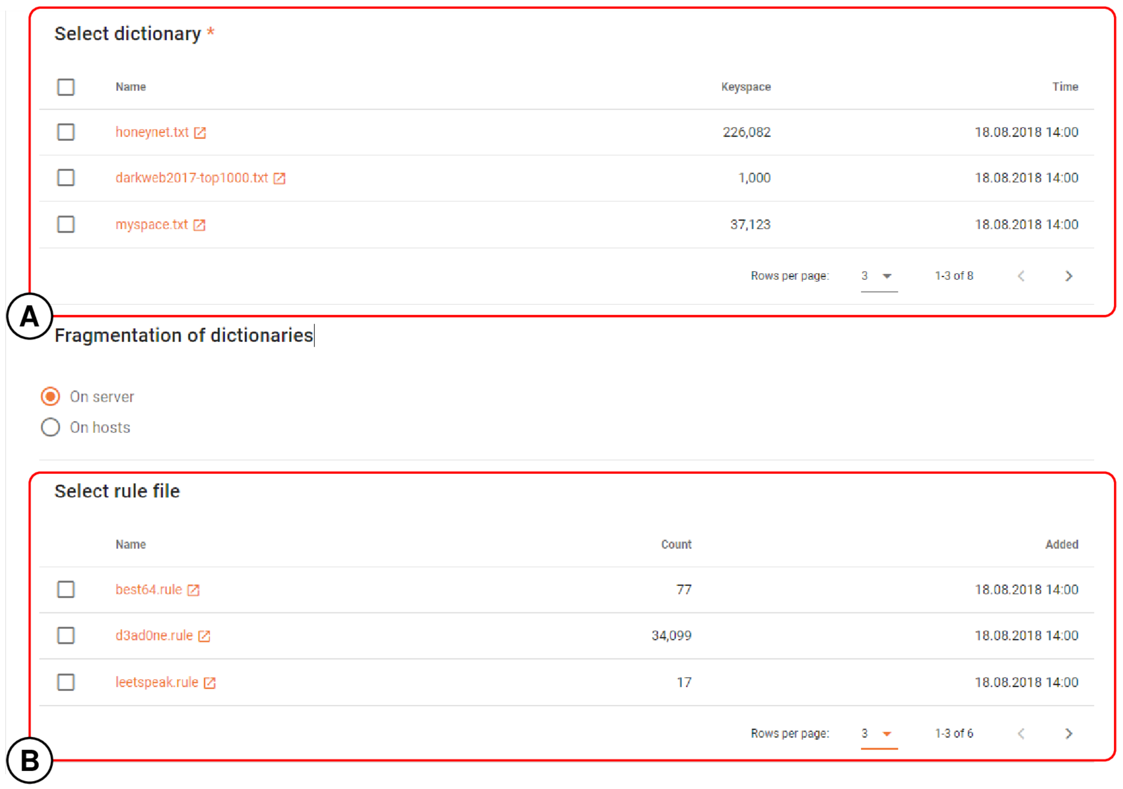
When cracking passwords, the way how the set of candidate passwords to be checked is generated—the attack mode—is the most important and most complex choice a user makes. It determines whether passwords are successfully cracked, as the set must contain the secret password. It also determines the length of the cracking process, as a large set of candidate passwords takes a longer time to iterate through. Fitcrack supports six different attack modes. This section explains how the different attack modes work and how they are configured from the Webadmin front end.

2.2.1 Dictionary Attack

In this type of attack, a text file called a *dictionary* is used. Each line of text in the dictionary represents a password to be checked. More than one dictionary can be chosen to be used—in that case, the selected dictionary files are treated as one dictionary created by concatenating all the individual dictionaries. Fitcrack also supports a feature called “password mangling”. This feature allows extending the keyspace by applying mangling rules to each individual password, creating modified versions of the original passwords. Mangling rules specify actions such as converting letters from uppercase to lowercase, reversing strings, substituting characters, etc. These rules are specified in the rule language used by hashcat[4] and are passed to Fitcrack in the form of *rule files*. A rule file is a text file containing a rule on each line. Fitcrack takes each rule from the rule file and applies it to each password in the dictionary. As an example, if the selected dictionary contains two passwords, `foo` and `bar`, and the selected rule file contains two rules, capitalise the string and reverse the string, the resulting keyspace to be used for cracking contains four candidate passwords, namely `Foo`, `oof`, `Bar`, and `rab`.

In the Webadmin front-end (Fig. 2.9), the user selects the dictionaries they want to use by ticking the check boxes next to their name (A). The Fitcrack server keeps a repository

[4]`https://hashcat.net/wiki/doku.php?id=rule_based_attack`



Select dictionary *

☐	Name	Keyspace	Time
☐	honeynet.txt	226,082	18.08.2018 14:00
☐	darkweb2017-top1000.txt	1,000	18.08.2018 14:00
☐	myspace.txt	37,123	18.08.2018 14:00

Rows per page: 3 1-3 of 8

Fragmentation of dictionaries

[✓ On server]
On hosts

Select rule file

	Name	Count	Added
☐	best64.rule	77	18.08.2018 14:00
☐	d3ad0ne.rule	34,099	18.08.2018 14:00
☐	leetspeak.rule	17	18.08.2018 14:00

Rows per page: 3 1-3 of 6

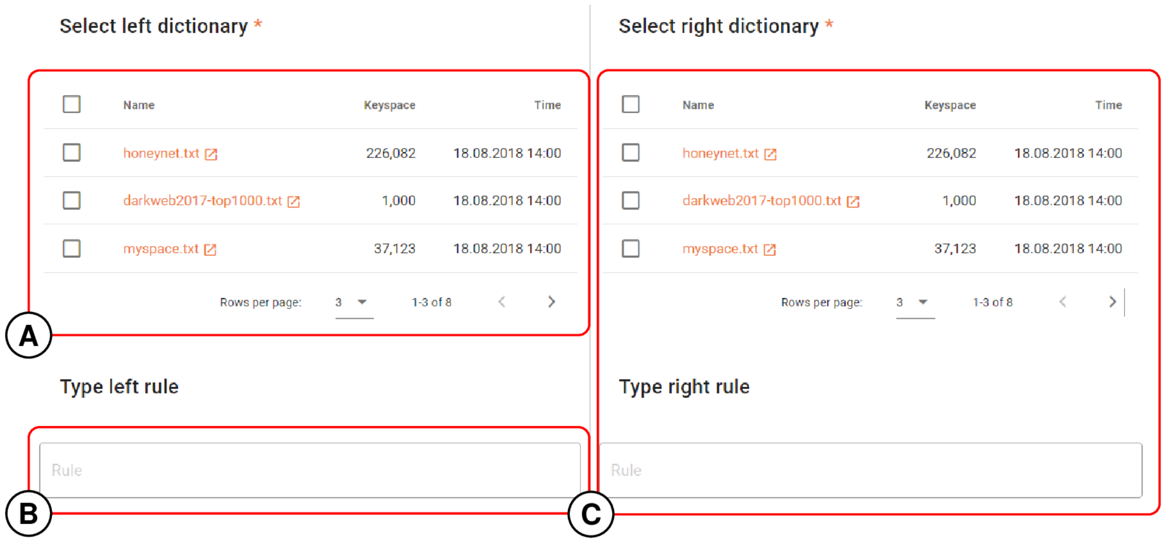
Figure 2.9: The Fitcrack Webadmin interface for configuring a dictionary attack — dictionary selection (A), ruleset selection (B)

of dictionaries, which are shown here. Dictionaries can be added or removed from this repository from the library-management pages in Webadmin (Section 2.3). Similarly, the user can select which (if any) rule files they want to use (B).

2.2.2 Combination Attack

The combination attack is in many ways similar to the dictionary attack discussed previously. It also works with dictionaries; however, it uses two dictionaries at once and combines them. The dictionaries are referred to as the “left” and “right” dictionaries. The attack works by taking every individual password from the left dictionary and concatenating it with every individual password from the right dictionary. So for example, if the left dictionary contains the passwords `foo` and `alice`, and the right dictionary contains the passwords `bar` and `bob`, the resulting keyspace contains four passwords, they being `foobar`, `foobob`, `alicebar`, and `alicebob`. Additionally, a single mangling rule (Section 2.2.1) can be applied to the left and right dictionaries (one per each). The passwords in the dictionaries will be then mangled before being used for the concatenation.

The Webadmin front-end (Fig. 2.10) then presents the configuration options like so: The dictionaries files for inclusion in the left dictionary are presented to the user as a list, where the user selects their desired dictionaries by ticking their respective check boxes (A). The user can then optionally input a mangling rule to be used on the left dictionary by typing it into a text box (B). The options for the right dictionary are then the same as for the left and are mirrored next to the left-dictionary options (C).


Select left dictionary *

	Name	Keyspace	Time
☐	honeynet.txt	226,082	18.08.2018 14:00
☐	darkweb2017-top1000.txt	1,000	18.08.2018 14:00
☐	myspace.txt	37,123	18.08.2018 14:00

Rows per page: 3 1-3 of 8

Select right dictionary *

	Name	Keyspace	Time
☐	honeynet.txt	226,082	18.08.2018 14:00
☐	darkweb2017-top1000.txt	1,000	18.08.2018 14:00
☐	myspace.txt	37,123	18.08.2018 14:00

Rows per page: 3 1-3 of 8

Type left rule

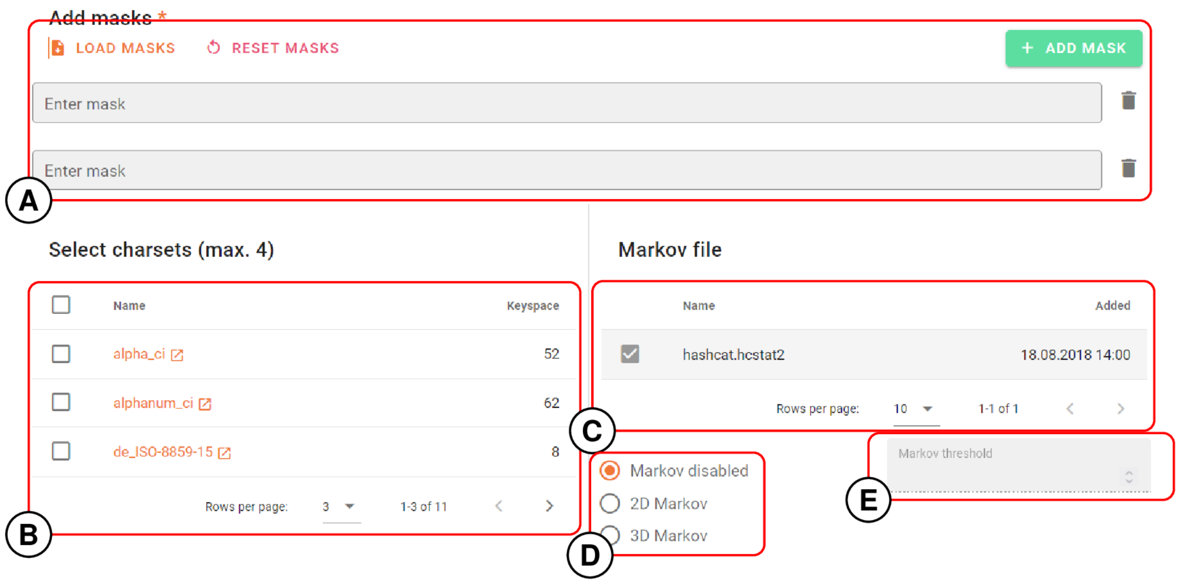
Rule

Type right rule

Rule

Figure 2.10: The Fitcrack Webadmin interface for configuring a combination attack — dictionary-file selection for the left dictionary (A), rule input for the left dictionary (B), mirrored configuration for the right dictionary (C)

2.2.3 Brute-force Attack


Add masks *

LOAD MASKS RESET MASKS + ADD MASK

Enter mask

Enter mask

Select charsets (max. 4)

	Name	Keyspace
☐	alpha_ci	52
☐	alphanum_ci	62
☐	de_ISO-8859-15	8

Rows per page: 3 1-3 of 11

Markov file

	Name	Added
✓	hashcat.hcstat2	18.08.2018 14:00

Rows per page: 10 1-1 of 1

[✓ Markov disabled] 2D Markov 3D Markov

Markov threshold

Figure 2.11: The Fitcrack Webadmin interface for configuring a brute-force attack — mask editor (A), selection of custom character sets (B), selection of Markov statistics file (C), selection of Markov-chain mode (D), input for Markov threshold (E)

The brute-force attack works by trying out every password that matches a *password mask*. A password mask is a specification of the valid values for each character position in a password. A character in this context is defined strictly as a one-byte value.

Let us look at an example, the mask `?u?l?l?d`. In the mask, the substitution sequences `?u`, `?l`, and `?d` each represent an uppercase alphabetical character (A–Z), lowercase alphabetical character (a–z), and decimal digit (0–9) respectively. The valid passwords for our mask are thus the strings `Aaa0`, `Aaa1`, ..., `Zzz8`, `Zzz9`. It is also in this lexicographical order that the password candidates are generated by default. The sets of characters that can be put in place of specific substitution sequences are called the *charsets* of those substitution sequences (so `?u` represents the charset of uppercase characters). In addition to the three charsets hereto shown, there exist eight default charsets in total.[5] Moreover, there exists the option to specify custom charsets: up to four charset files can be provided, and the characters therefrom can be used in a password with the substitution sequences `?1`, `?2`, `?3`, and `?4`, representing the first to fourth custom charset respectively. Aside from substitution sequences of the form question mark followed by a single-characer charset specifier, it is also possible to include static letters in the mask. These represent characters that are unchanging. In the mask, they are indicated by the literal character they represent.[6] So the mask `?dA?d` represents all passwords that are in the form of a decimal digit, followed by the letter A, followed by another decimal digit. Multiple masks can also be used concurrently; then the resulting keyspace is formed by all the passwords generated by all provided masks.

There is a big downside with the brute-force approach as described: most candidate passwords generated therefrom are not very likely to be used by most people. However, Fitcrack supports a feature to alleviate this drawback: Markov chains. The principle behind this feature is that some letter combinations are more likely to appear in passwords than others. For example, the letter B is more likely to be followed by the letter Y (e.g. when forming the word "by") than the letter Q (there are few English words that contain the substring "bq"). This is called the *2D Markov* mode. There also exists the *3D Markov* mode, which in addition to using the information about the previous character uses the position in the password to judge character probabilities (e.g. uppercase characters are more likely to appear at the beginning of passwords, whereas numbers are likely to be at the end). Fitcrack can then generate passwords not in the lexicographical order, but in the order that the characters are most likely to appear relative to each other, thus potentially cracking passwords in a faster time. To use this feature, the user needs to supply a "Markov statistics file", which contains information about the relative character probabilities that are to be used. However, though this feature generates likelier candidate passwords first, it does not prevent generating unlikely candidate passwords. If the user wishes to limit the keyspace to likelier passwords, they can set a *Markov threshold* value. The Markov threshold is a positive integer value that limits the generation of candidate passwords by only using the N likeliest character combinations as defined by the Markov statistics file, where N is the Markov threshold value. For example, suppose the Markov statistics file defines the four likeliest letters after the letter A to be T, Y, N, and W, and suppose the Markov threshold is set to three, then a sequence of thus-generated passwords may look like so: `mat`, `may`, `man`, `men`. We can see that after the word `mat` comes the word `may` because the Markov statistics file defines T to be the likeliest letter after A, followed by Y. Likewise this is true for `may` and `man`. However, `man` is not followed by `maw`, and that is because W is the fourth most likely letter to follow after the letter A, and the Markov threshold

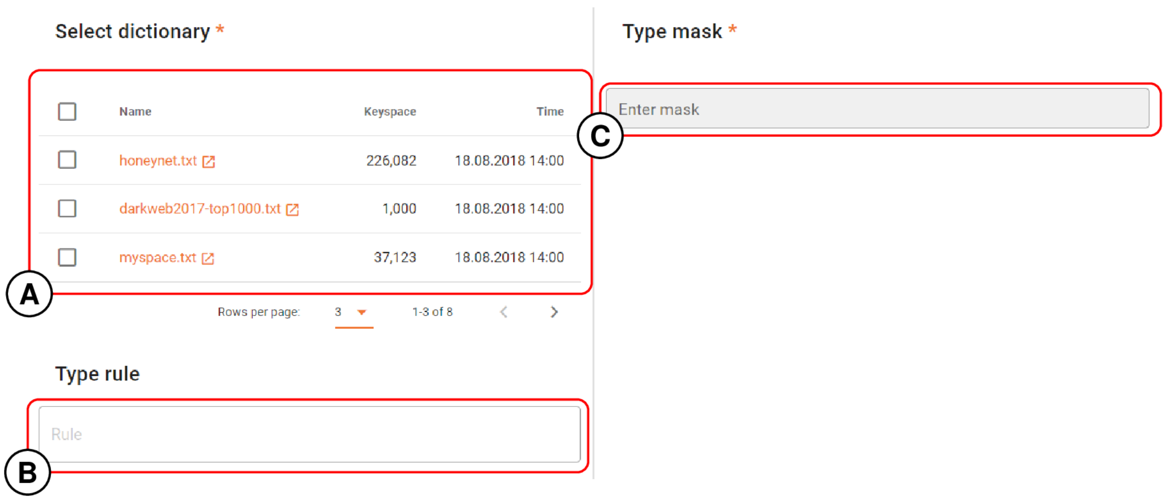
[5]In addition to the three charsets shown, there exist:
`?h` — lowercase hexadecimal digits (0–f)
`?H` — uppercase hexadecimal digits (0–F)
`?s` — special characters (`[space]!„#$:;<=>?@[]^_‘{|}~`)
`?a` — all standard ASCII characters
`?b` — binary; all 8-bit binary values

[6]To include the static letter representing a question mark, it has to be escaped like so `??` as otherwise there would be no way to differentiates between the static letter `?` and the start of substitution sequence.

value of three tells us we only care about the three most likely characters that follow each individual character, so in the next iteration the letter A in the second position in the word is modified (changing A to E), and the last position in the word is set to the character most likely to follow after the new letter (E is most likely followed by N), giving us the next generated password of `men`. Of course, this Markov-chain approach depends on the assumption that the searched passwords contain predictable patterns that can be modelled by Markov chains. But if the user can make such an assumption, this feature can increase the speed of password cracking using the brute-force attack.

In the context of the Webadmin front-end (Fig. 2.11), the user enters the masks to be used one by one into individual text boxes, or they load masks from a hashcat mask file, which can then be modified as if the user entered them one by one (A). The user can select up to four custom charsets by checking check boxes next to the charset names they want to use (B). If the user wants to generate candidate passwords using the Markov-chain–based order, they can select which mode they want to use (D), select the Markov statistics file to be used (C), and also optionally set a Markov threshold value (E).

2.2.4 Hybrid Attack



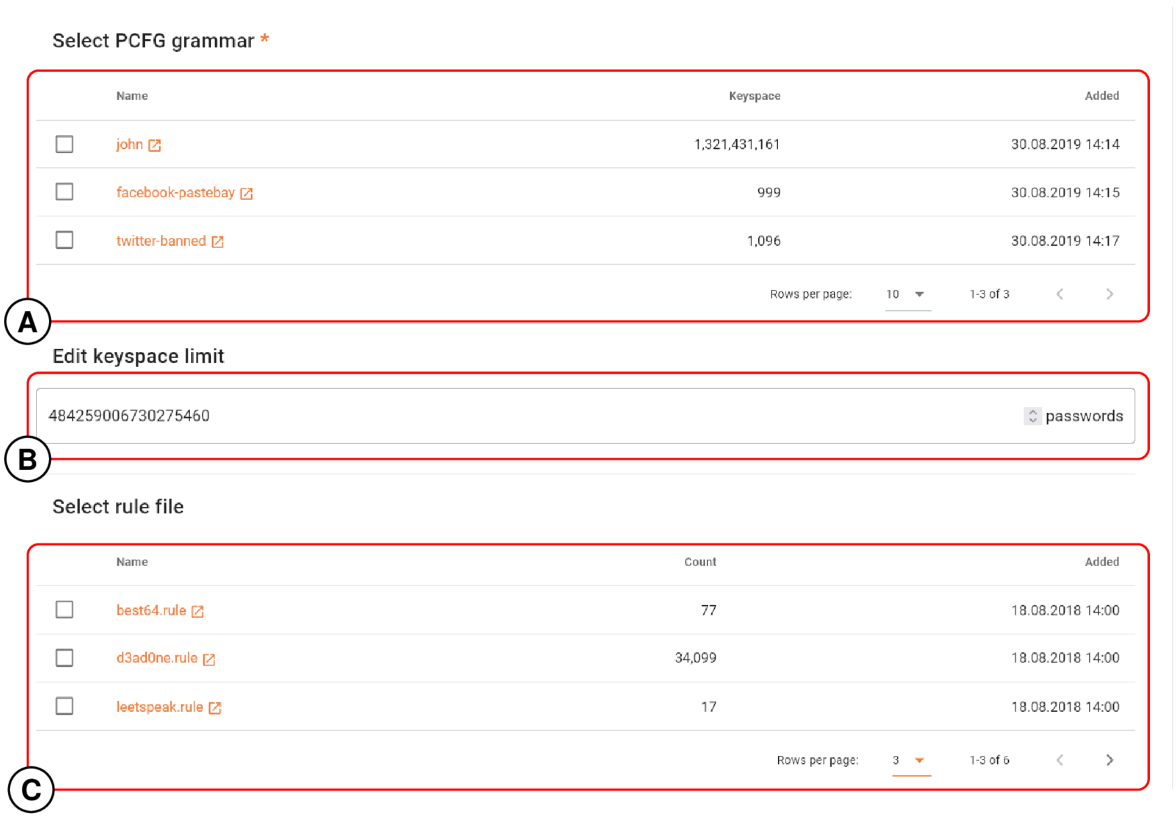
	Name	Keyspace	Time
☐	honeynet.txt	226,082	18.08.2018 14:00
☐	darkweb2017-top1000.txt	1,000	18.08.2018 14:00
☐	myspace.txt	37,123	18.08.2018 14:00

Figure 2.12: The Fitcrack Webadmin interface for configuring a hybrid wordlist and mask attack — dictionary-file selection for the left dictionary (A), rule input for the left dictionary (B), password-mask input for the mask to generate right-side passwords for concatenation

The hybrid attack is very similar to the combination attack (Section 2.2.2), but instead of concatenating two passwords from two dictionaries, passwords from a dictionary are concatenated with passwords generated using the brute-force method (Section 2.2.3). It is also possible to concatenate the other way around—a password generated using the brute-force approach with a password from a dictionary. Similarly to the combination attack, only one mangling rule (Section 2.2.1) can be applied to the dictionary. There are also restrictions placed on the candidate passwords generated with the brute-force approach: only a single simple password mask may be used, i.e. no custom charsets may be used, and the generation order is strictly lexicographic.

From the Webadmin front-end perspective (Fig. 2.12), configuring—in this case a hybrid wordlist and mask attack—is very similar to that of the combination attack. The user selects the dictionary to be used for the left side (A) and optionally types a mangling rule to be used with it (B); then they type in a password mask for the right side (C). The UI for the opposite hybrid mask and wordlist attack is nearly identical, with the dictionary- and mask-selection UI sections swapping places.

2.2.5 PCFG Attack


Select PCFG grammar *

	Name	Keyspace	Added
☐	john	1,321,431,161	30.08.2019 14:14
☐	facebook-pastebay	999	30.08.2019 14:15
☐	twitter-banned	1,096	30.08.2019 14:17

Rows per page: 10 1-3 of 3

A

Edit keyspace limit

484259006730275460 passwords

B

Select rule file

	Name	Count	Added
☐	best64.rule	77	18.08.2018 14:00
☐	d3ad0ne.rule	34,099	18.08.2018 14:00
☐	leetspeak.rule	17	18.08.2018 14:00

Rows per page: 3 1-3 of 6

C

Figure 2.13: The Fitcrack Webadmin interface for configuring a PCFG attack — selection of grammar to use (A), input for keyspace-size limit (B), selection of mangling ruleset (C)

PCFG is an initialism for "probabilistic context-free grammar". The attack works by analysing a set of pre-existing passwords and deriving a probabilistic context-free grammar from them. Since the grammar is based on an existing set of passwords, it should capture the patterns of password creation present in the original set of passwords. For example, suppose that the set of passwords that is analysed contains passwords in the form of "word number word" (e.g. `foo 42 bar` and `alice 16 bob`). In the analysis process, this pattern is picked up and introduced into the grammar. After the analysis process is complete and a PCFG is generated, the attack generates all passwords that are valid according to the grammar. This set of generated passwords should be larger than the original set because the generated grammar should generalise the rules and create new valid combinations. So for example, the "word number word" should get picked up and password candidates like `alice 42 foo` should appear in the resulting generated keyspace despite not being in the original set. In the analysis process, the relative occurrences of substrings is noted, and this information is used to add probabilities to the terms in the generated grammar. This means that when password candidates are generated from the created grammar, password candidates that are most probable to be used are generated first. The PCFG attack was originally proposed by Weir et al. [17].

Fitcrack has the option to generate PCFGs from dictionaries. The resulting PCFGs are then stored for later use.[7] A stored PCFG can then be selected and, in the attack, candidate passwords are generated therefrom and used to try to crack the requested hashes. Fitcrack also offers a way to limit how many passwords are generated from the PCFG. Mangling (Section 2.2.1) can also be applied to the generated passwords to expand the keyspace.

When it comes to setting up a PCFG attack in the Webadmin front-end (Fig. 2.13), the user selects a single stored PCFG to be used as a base for generating passwords (A). They can also limit the number of candidate passwords by setting the maximum number of passwords to generate (B), and they can select a single mangling ruleset file to be applied to the candidate-password set generated by the PCFG (C). If the user wishes to create a new PCFG, they can either import one or generate one from a dictionary through the cracking-asset library (Section 2.3) in Fitcrack Webadmin.

2.2.6 PRINCE Attack

PRINCE stands for "PRobability INfinite Chained Elements". It is a modern password-generation algorithm designed by Jens Steube [14]. The algorithm works by concatenating several strings together. All strings are taken from a single dictionary, and the order in which the strings are concatenated is chosen depending on the relevance (occurrence frequency) of the strings. So for example, if we use the PRINCE algorithm with a dictionary consisting of three words `foo`, `bar`, and `nill`, the PRINCE algorithm will generate passwords such as `foo`, `foobar`, `nillfoobarbar` etc. However, since this leads to generating an infinite keyspace, the algorithm takes also as input a set of limits: a minimum and maximum length for the password as a whole, and a minimum and a maximum number of strings that are concatenated. This limits the size of the keyspace to a finite number. Another configuration setting taken by the PRINCE algorithm is whether to use case permutation; if yes, the PRINCE algorithm can change the capitalisation of the first character of a word from the input dictionary. I.e. if the dictionary contains the word `foo`, the PRINCE algorithm behaves as if there were two words, `foo` and `Foo` in the dictionary. This allows the PRINCE algorithm to generate common camel-case–style passwords (e.g. `myPasswordFoo`) without needing to duplicate words in the input dictionary. Additionally, a separate total limit on the number of candidate passwords generated can be imposed. This is separate from the password-length and element-count limits—if the total limit is hit while the PRINCE algorithm can still generate new password candidates, then generation is prematurely ended (and vice versa if the PRINCE algorithm terminates before the limit is hit). Finally, a check for duplicate passwords can be enabled—given how the algorithm works, it is possible that a given string can be generated independently several times by concatenating various numbers of input words. Fitcrack also allows to use one ruleset file for mangling (Section 2.2.1) the words in the input dictionary, and also specifically for this attack mode only, Fitcrack allows the generation of random mangling rules to be used with the input dictionary. The usage of random rule generation and a ruleset file cannot be combined.

When looking at the PRINCE-attack configuration UI in Webadmin (Fig. 2.14), the discussed options are laid as so: The user can select several dictionary files to be used as

[7]Do note that the PCFGs are stored, not sets of candidate passwords generated by them, as the size of those sets can be massive and impractical.

the input dictionary (A). The user can set the limits of the PRINCE algorithm by typing in the minimum and maximum values (D) and also the total keyspace limit (E). One can also enable or disable the options to use case permutation (C) and to check and eliminate potential duplicates (B). Finally, the user can select one mangling ruleset file to apply to the input dictionary (F) or input the desired number of mangling rules to be randomly generated (G), but not both at once.

2.3 Library Management

Another major part of Fitcrack Webadmin is the management of cracking assets, files required for cracking jobs: dictionaries, PCFGs, rulesets, charsets, mask files, and Markov statistics files. Collectively, this is called the asset library.

In this section, we step through the actions a user takes to manage cracking assets. Although there are 6 different types of cracking assets, the UI for managing them is very similar. The actions are first shown for managing rule files (Fig. 2.15). Followed are only the differences for other types of cracking assets, where they exist.

For cracking assets that are stored on the Fitcrack server, two actions are always available to be performed: downloading the asset file and deleting it.

To download a cracking-asset file, the user clicks the download button (Fig. 2.15C). After clicking, a regular file download is initiated.

To delete a cracking asset, the user clicks the delete button (Fig. 2.15D). After clicking it, a pop-up dialog box appears, asking the user if they are sure. The user can then either press the enter key or click on the “Yes” button to confirm the deletion. A snackbar notification then appears to inform the user that the deletion was successful (or that an error occurred).

These two actions are identical for all types of cracking assets.

To upload a new cracking asset, the user first clicks on the file input (Fig. 2.15A) and selects a file to be uploaded. Then the user confirms the upload by clicking the “Upload” button (Fig. 2.15B) to confirm the upload. A snackbar notification then appears to inform the user that the upload was successful (or that an error occurred). This sequence of actions is the same for ruleset, charset, and mask-file management.

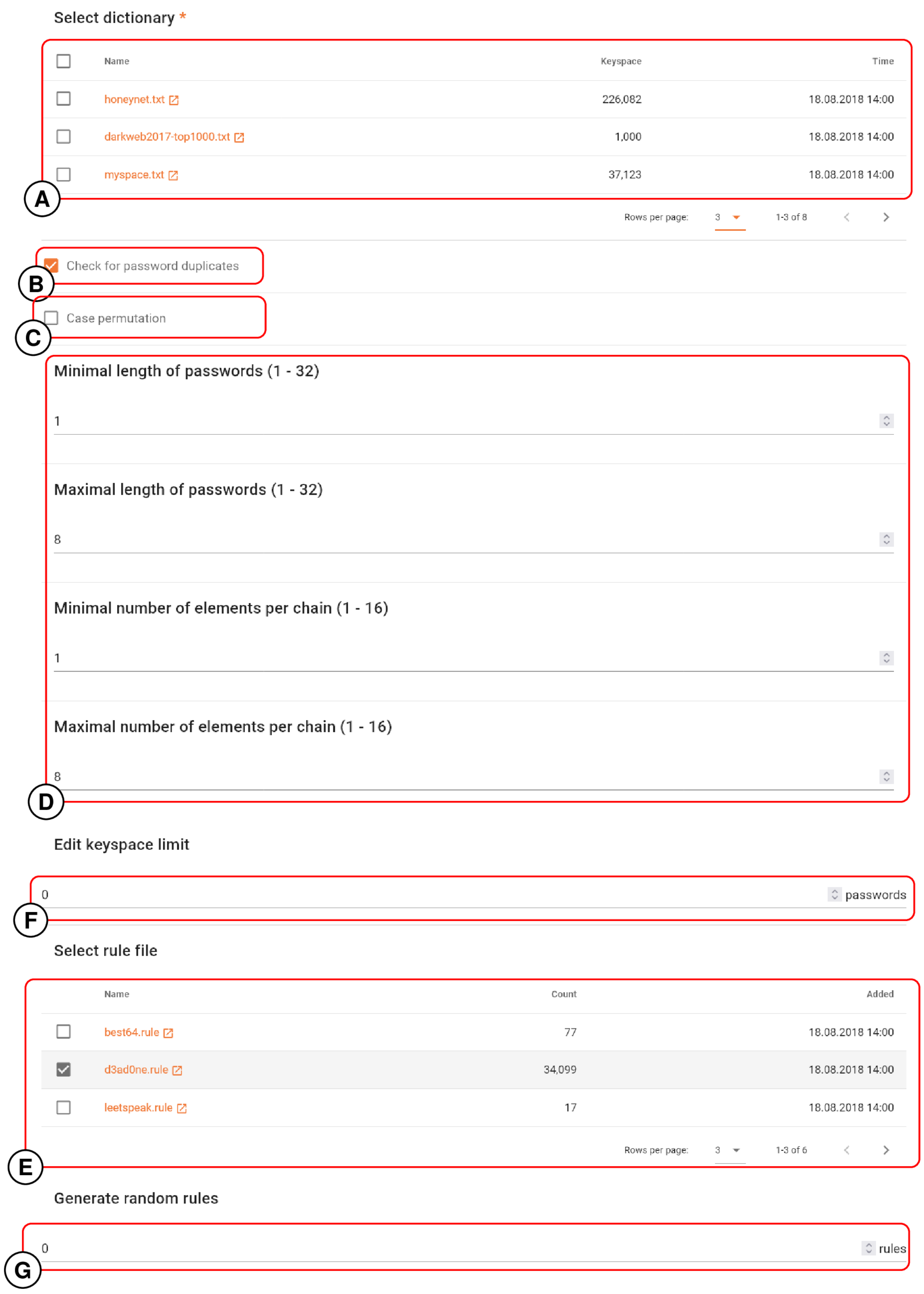
For the management of dictionaries, PCFGs, and Markov statistics files, the way to upload new files is different. The example is shown on Markov statistics file management, but there virtually no differences for the three types of assets. First, the user clicks the “Add New” button (Fig. 2.16). This opens up a dialog box (Fig. 2.17).

If the user wishes to upload a new file, the user can perform the same steps as described above for the other types of cracking assets, using the same upload UI (Fig. 2.17B) as in Fig. 2.15.

The user can also create Markov statistics files by generating them from dictionaries. To do that, the user first navigates to the “Make from Dictionary” tab (Fig. 2.17A) in the dialog box. After switching to the other tab (Fig. 2.18), the user can select which dictionary they want to use for generating (Fig. 2.18A). They can then optionally set the name the newly-generated Markov statistics file should use (Fig. 2.18B). And finally, the user confirms their intent to generate the Markov statistics file by clicking the confirm button (Fig. 2.18C). After some time, the dialog closes and a snackbar notification appears to inform the user that the upload was successful (or that an error occurred).

Similarly, PCFGs can be created from dictionaries.

Finally, there is a special-case: the management of dictionary files. When managing dictionary files, there are some additions (Fig. 2.19) to the regular UI. There are two additional check boxes, one to select whether to sort the uploaded file by password length and one whether to treat the uploaded file as a dictionary containing password stored in hexadecimal format. The user selects these options before clicking “Upload New” and proceeding as described above. There is also an additional way of uploading files—importing from a special folder on the Fitcrack server by clicking the “Add From Server” button.



Select dictionary *

☐	Name	Keyspace	Time
☐	honeynet.txt	226,082	18.08.2018 14:00
☐	darkweb2017-top1000.txt	1,000	18.08.2018 14:00
☐	myspace.txt	37,123	18.08.2018 14:00

Rows per page: 3 1-3 of 8

☑ Check for password duplicates

☐ Case permutation

Minimal length of passwords (1 - 32)

1

Maximal length of passwords (1 - 32)

8

Minimal number of elements per chain (1 - 16)

1

Maximal number of elements per chain (1 - 16)

8

Edit keyspace limit

0 passwords

Select rule file

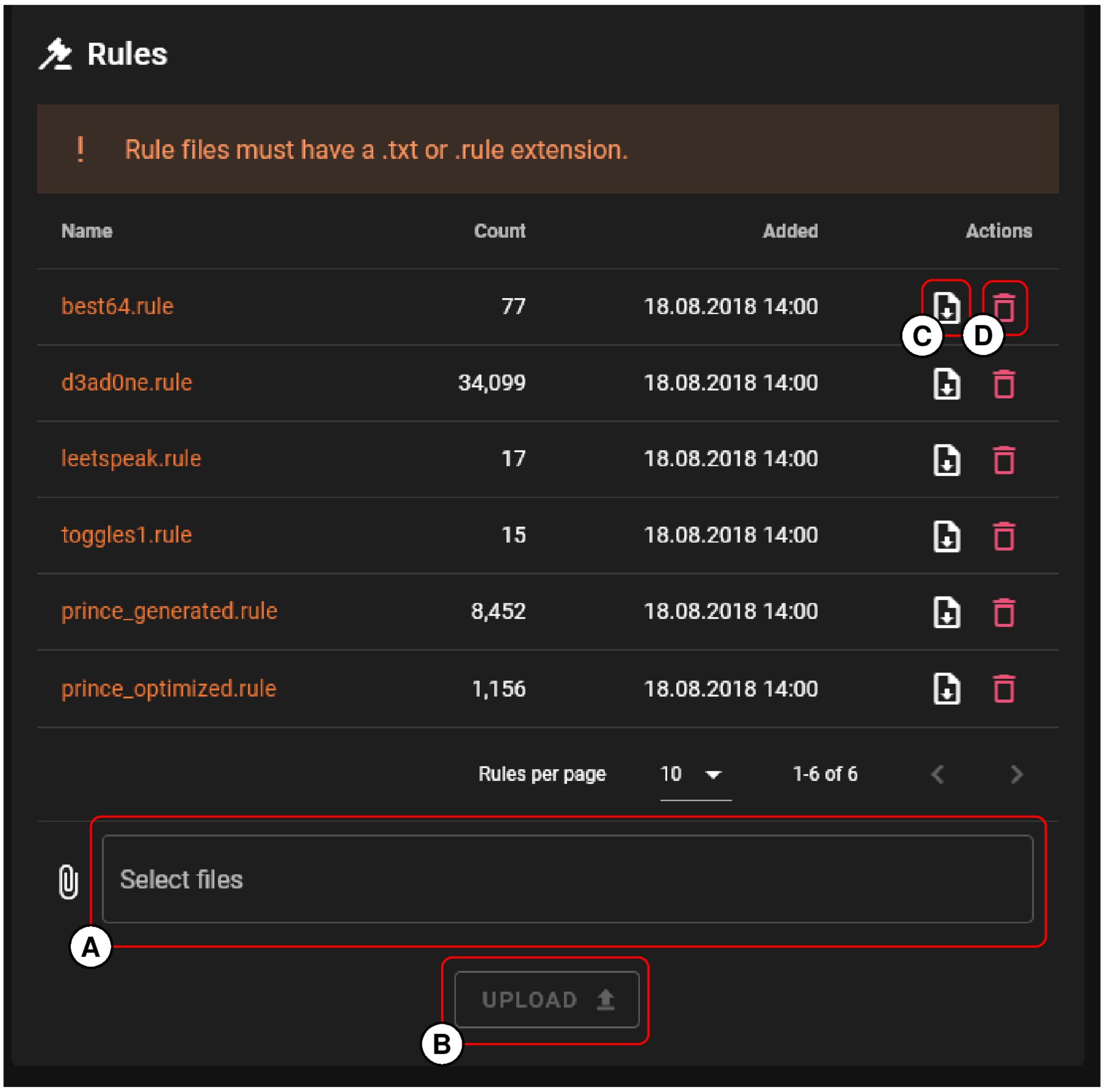
	Name	Count	Added
☐	best64.rule	77	18.08.2018 14:00
☑	d3ad0ne.rule	34,099	18.08.2018 14:00
☐	leetspeak.rule	17	18.08.2018 14:00

Rows per page: 3 1-3 of 6

Generate random rules

0 rules

Figure 2.14: The Fitcrack Webadmin interface for configuring a PRINCE attack — selection of dictionary-files to create input dictionaries (A), option for checking and eliminating of duplicates (B), option whether to use case permutation (C), inputs for setting the PRINCE generation limits (D), input for limiting the size of the keyspace as a whole (E), selection of a single mangling ruleset to be used with the input dictionary (F), input for the number of randomly generated mangling rules (G)

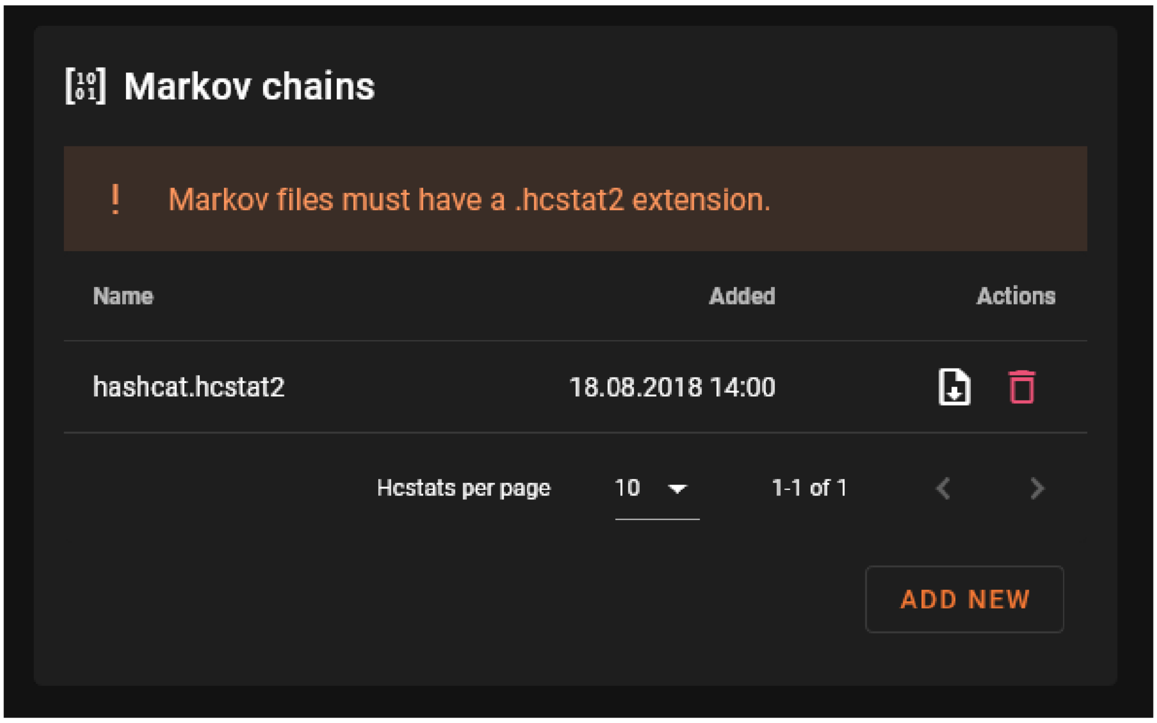


Rules

! Rule files must have a .txt or .rule extension.

Name	Count	Added	Actions
best64.rule	77	18.08.2018 14:00	
d3ad0ne.rule	34,099	18.08.2018 14:00	
leetspeak.rule	17	18.08.2018 14:00	
toggles1.rule	15	18.08.2018 14:00	
prince_generated.rule	8,452	18.08.2018 14:00	
prince_optimized.rule	1,156	18.08.2018 14:00	

Figure 2.15: The Fitcrack Webadmin interface for managing rule files — file input for selecting file to be uploaded (A), upload button to confirm and perform a file upload (B), button to download a stored rule file (C), button to delete a stored rule file (D). The interface is virtually identical to the one for managing charsets and mask files.



Markov chains

! Markov files must have a .hcstat2 extension.

Name	Added	Actions
hashcat.hcstat2	18.08.2018 14:00	

Hcstats per page 10 1-1 of 1

ADD NEW

Figure 2.16: The Fitcrack Webadmin interface for managing Markov statistics files. The interface is mostly identical to the one for managing dictionaries and PCFGs.

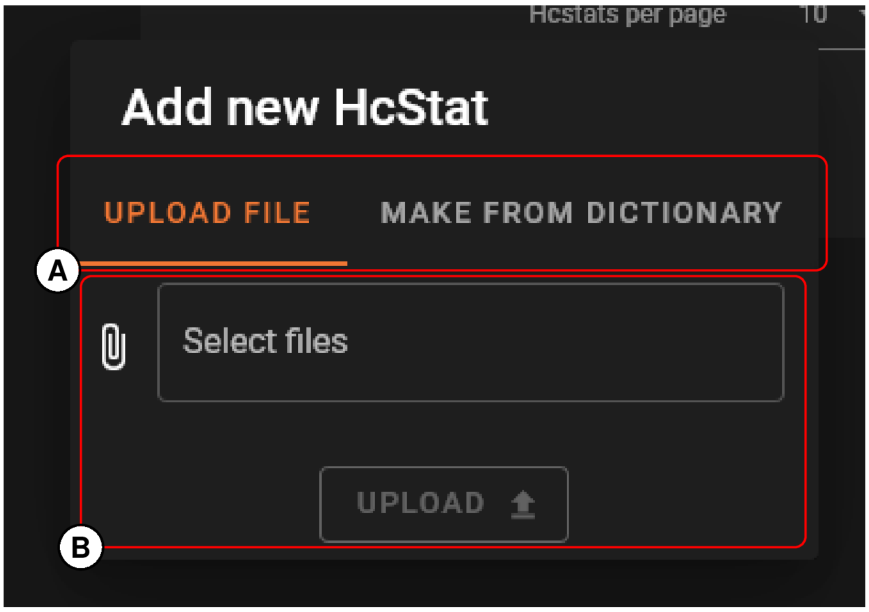
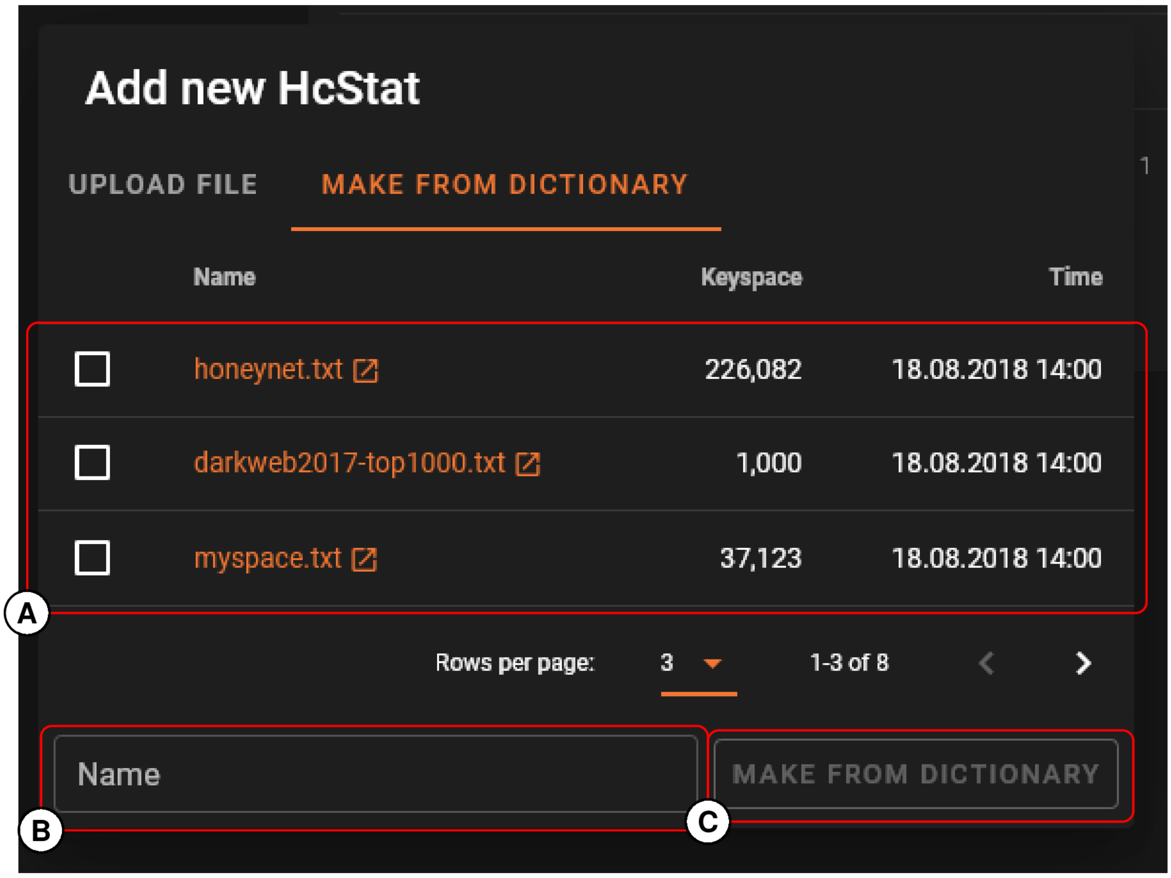


Figure 2.17: The dialog box for uploading a Markov statistics file; shown is the tab for uploading files — tabs for switching between file-upload and make-from-dictionary modes (A), file-upload UI (B). Note the upload UI is the same as in Fig. 2.15



Add new HcStat

UPLOAD FILE | MAKE FROM DICTIONARY

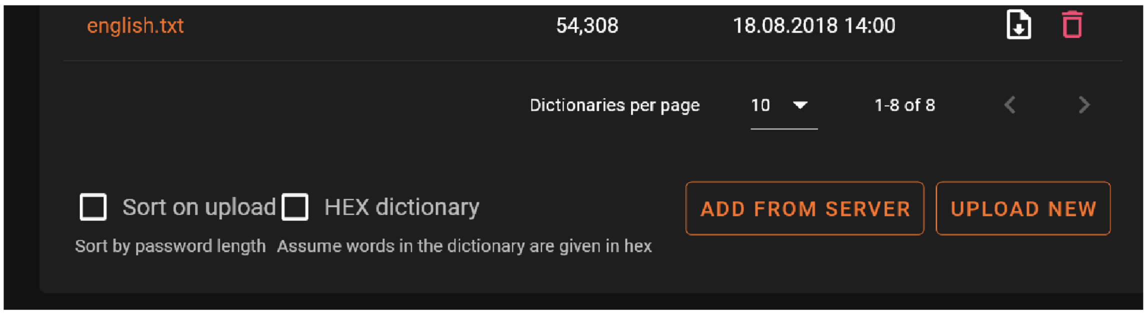
	Name	Keyspace	Time
☐	honeynet.txt	226,082	18.08.2018 14:00
☐	darkweb2017-top1000.txt	1,000	18.08.2018 14:00
☐	myspace.txt	37,123	18.08.2018 14:00

Rows per page: 3 1-3 of 8

Name

MAKE FROM DICTIONARY

Figure 2.18: The dialog box for uploading Markov statistics file; shown is the tab for creating Markov statistics file from dictionaries — selection table for the dictionary from which to create the Markov-statistics file (A), name field to set the name for the generated Markov statistics file (B), confirmation button (C)



english.txt	54,308	18.08.2018 14:00

Dictionaries per page 10 1-8 of 8

☐ Sort on upload ☐ HEX dictionary

Sort by password length Assume words in the dictionary are given in hex

ADD FROM SERVER UPLOAD NEW

Figure 2.19: Additional options that exist for the management of dictionary files.

Chapter 3

Methods for Testing Web Applications

This chapter discusses several methods for creating tests for web applications.

A naive way of writing tests is to lay out a sequence of actions to be performed, such as clicking a button, typing some text, and scrolling the page. This approach is called *capture–replay*, as in we are capturing actions to be replayed later. While this approach is easy to understand and does not even require writing code by virtue of using recording tools such as Selenium IDE, it brings forth some problems.

Suppose we have a test suite and that all tests start by logging in as a user. If we use a capture–replay approach, all tests start with the same piece of code that logs the user in. Moreover, the code contains very specific actions such as "click the element with id 'foo'". If the website changes its layout, all tests break and all tests need to be modified to be fixed.

Martin Fowler proposed a solution to this problem. Originally called the *window model*, it is now more commonly known as the *page-object model* [2]. In the page-object model, every web page or significant part of a web page is modelled as a page object. A page object encapsulates the how of interacting with the web page. It contains the logic of how website elements are located in the DOM tree, and it exposes interactions with the web page in the form of methods. Crucially, the interactions are modelled from the end user's point of view. And thus the page-object model achieves

1. deduplication of code, repeat actions can be bundled into methods and then reused;
2. separation of concerns, code that interacts with the UI is separate from the tests that interact with the app.

We can demonstrate the difference between the approaches on the login page of Fitcrack (Fig. 2.1). Let us consider that we want to test two functionalities of this page, a successful login attempt and a failed login attempt. The tests are almost the same—they both fill in a username and password and click the "SIGN IN" button. What is different is what they check for after clicking the button—testing for successful login requires checking that we are on the home page and the correct user is shown as logged in; testing for a failed login should cause an error message to appear.

In the capture–replay approach (Listing 3.1), the tests consist of a sequence of literal steps taken.

```
1  class SuccessFullLogin(unittest.TestCase):
2      def test(self):
3          self.driver.get("https://live.fitcrack.cz")
4          self.driver.find_element_by_id("input-20").clear()
5          self.driver.find_element_by_id("input-20").send_keys("Alice")
6          self.driver.find_element_by_id("input-23").clear()
7          self.driver.find_element_by_id("input-23").send_keys("foo")
8          self.driver.find_element_by_xpath('//button[@type="submit"]').click()
9          WebDriverWait(self.driver, timeout=15).until(lambda d: d.find_element_by_tag_name("h1"))
10         assert "Alice" in self.driver.find_element_by_tag_name("h1")
11 class FailedLoginTest(unittest.TestCase):
12     def test(self):
13         self.driver.get("https://live.fitcrack.cz")
14         self.driver.find_element_by_id("input-20").clear()
15         self.driver.find_element_by_id("input-20").send_keys("wrong_login")
16         self.driver.find_element_by_id("input-23").clear()
17         self.driver.find_element_by_id("input-23").send_keys("foo")
18         self.driver.find_element_by_xpath('//button[@type="submit"]').click()
19         assert "User not found" in self.driver.find_element_by_class_name("v-alert__content").text
```

Listing 3.1: Login-page tests utilising the capture–replay approach

Let us first take a look at the successful-login test: On Line 3, we navigate to the login page. Then on lines 4–7, we input the login details into the login form. Line 8 clicks the submit button. Line 9 then waits until the homepage is loaded. Finally, Line 10 checks to see if our username appears in the welcome text on the homepage.

Observe now the failed-login test: We can see that Lines 13–18 are identical to Lines 3–8 in the successful-login test save for changes in the values typed into the text fields. What differs is Line 19: instead of checking that we are on the homepage, we check that an error message appears. This is an example of the duplication described heretofore.

Aside from the duplication of code, there is the problem of test fragility and maintenance: Suppose that in a new version of Fitcrack the ID of the username field changes from `input-20` to `input-username`. In that case, all tests break and all tests need to replace their locators to refer to the new id. In this case, it is a rather trivial amount of four lines needing to be amended; however, in large test suites, the number of lines needing to be changed can be large.

```
1  class SuccessFullLogin(unittest.TestCase):
2      def test(self):
3          loginPage = LoginPage(self.driver).navigate()
4          homePage = loginPage.login("Alice","foo")
5          assert "Alice" in homePage.getWelcomeText()
6  class FailedLoginTest(unittest.TestCase):
7      def test(self):
8          loginPage = LoginPage(self.driver,PREFIX).load()
9          loginPage.login("wrong_login","42069")
10         assert "User not found" in loginPage.getErrorText()
```

Listing 3.2: Login-page tests utilising the page-object–model approach

Now we will look at the same tests implemented using the page-object model approach (Listing 3.2). We can see that the tests are much shorter. They do not contain explicit actions to the web browser but rather actions expressed closer to the natural language of an end user.

Let us look at the successful-login test: On Line 3, we construct a `LoginPage` object and use its `navigate` method to make the web browser load the Fitcrack login page. Line 4 then uses the `login` method to log the user in; the method returns a `HomePage` object representing the home page of Fitcrack. And finally, Line 5 uses the `getWelcomeText` method to see if our username is in the welcome text of the homepage.

The failed-login test is quite similar, with the difference in the assertion at the end: Line 10 checks for the existence of an error message.

```
 1 class LoginPage():
 2     def __init__(self,driver:WebDriver):
 3         self.driver = driver
 4         self.URL = "https://live.fitcrack.cz/"
 5     @property
 6     def username_field(self):
 7         return self.driver.find_element_by_id("input-20")
 8     @property
 9     def password_field(self):
10         return self.driver.find_element_by_id("input-23")
11     @property
12     def submit_button(self):
13         return self.driver.find_element_by_xpath('//button[@type="submit"]')
14     @property
15     def error_text(self):
16         return self.driver.find_element_by_class_name("v-alert__content")
17     def navigate(self):
18         self.driver.get(self.URL)
19         return self
20     def getErrorText(self):
21         return self.errorText.text
22     def login(self,username:str,password:str):
23         self.username.send_keys(username)
24         self.password_field.send_keys(password)
25         self.submit_button.click()
26         return HomePage(driver)
```

Listing 3.3: Class representing the login page

The tests shown made use of page objects. These have to be implemented by the programmer. The class representing the login page is shown in Listing 3.3. Lines 5–16 contain the definitions of properties representing objects of interest on the web page. Each property consists of one getter that contains a locator call locating the page element. These properties are not used directly in tests but rather they are used by the methods of the page object. Definitions of such methods are shown on Lines 17–26.

Now let us again consider the situation when the id of the username field changes: just like in the example with the capture–replay approach, all tests break; however, with the page-object model approach, only one line of code needs to be changed, and that is the `username_field` property definition on Line 7. The deduplication achieved with the page-object model reduces the amount of maintenance needed. Furthermore, the tests are written in a more domain-specific language, expressing actions the end user wants to take in a more natural language.

The advantages of the page-object model approach have been practically measured and demonstrated in real-life applications and studies. It has been shown that the creation of test suites using the page-object model approach takes in general a longer initial time, but the effort required to maintain the test suite is greatly reduced. For that reason, I decided to use this approach in the development of a test suite for Fitcrack [7, 8, 9, 10].

Chapter 4

Selenium Browser-Automation Project

Selenium is a collection of tools and libraries used for browser automation. Its development started in 2004 as an internal project at the company ThoughtWorks by Jason Huggins. At this time, Selenium was a set of JavaScript scripts used to test websites and a domain-specific language to define what actions Selenium shall perform. These scripts had to be included onto the websites that developers wanted to test. The company soon thereafter decided to open-source the tool, and in the coming years, many people contributed to Selenium, and the project branched out into creating several tools. The original approach of using JavaScript scripts manually injected onto web pages has since been abandoned [13, 15].

As of now the tools that are part of Selenium are:

- Selenium WebDriver, a solution to control web browsers programmatically in common languages such as Java, Python, and JavaScript;
- Selenium Grid, a server to enable distributed running of Selenium WebDriver scripts across multiple devices;
- Selenium IDE, a browser extension allowing recording and playback of actions taken on a website.

4.1 Selenium WebDriver

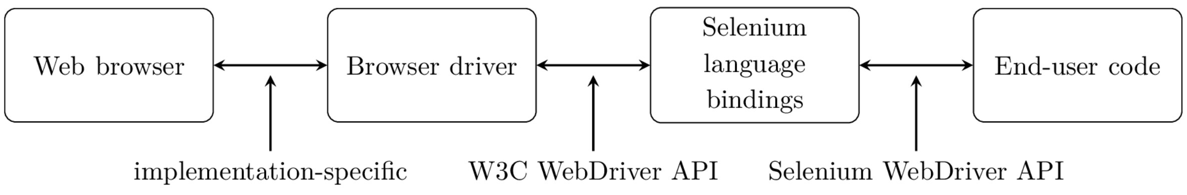


Figure 4.1: Overview of Selenium WebDriver components

Selenium WebDriver is the name given to a set of components and APIs that work together to let people write web-automation scripts. These are browser drivers, the W3C

WebDriver API, Selenium language bindings, and the Selenium WebDriver API. The interconnection of the components is shown in Fig. 4.1. In the following sections, the individual components are explained in detail, but for most uses, the following explanation is enough:

Selenium language bindings are libraries used by people to automate web-browser actions. They do not interface with web browsers directly; control of web browsers is performed by software called browser drivers. Selenium language bindings send commands to browser drivers using a unified API, which in turn control web browsers.

4.1.1 W3C WebDriver API

At the heart of Selenium WebDriver is the *W3C WebDriver API*, an API for programmatically controlling web browsers [16]. It exposes a set of commands for interacting with browsers as a set of REST endpoints. It is a W3C recommendation but was not always one: the API was created by the Selenium team and then subsequently standardised. It is different from the Selenium WebDriver API—the differences are described in Section 4.1.3. The provided commands include:

- navigating to URLs, refreshing the page, navigating backwards and forwards in the web-browser history;
- locating elements in the DOM tree in various ways, such as using CSS selectors, HTML IDs, XPath expressions etc.;
- interacting with those elements—clicking on them, typing into them if they are form inputs, querying whether an element is visible etc.;
- interacting in a more low-level way—clicking or scrolling on precise coordinates in a browser window, sending keystrokes to the browser etc.;
- executing arbitrary JavaScript.

This is not a complete enumeration of the commands the W3C WebDriver API offers; the commands listed are the most used and or important.

Actions that locate or interact with elements are the most commonly used by end users. The API specifies that these interactions be executed as if a human performed them. For example, this means if one sends a click-element command, the API specifies that the element must be visible and scrolled into view (and if not scrolled into view, the action should attempt to scroll the element into view), not obstructed by other elements, and be interactable (e.g. input elements must have the `disabled` attribute set), among many other smaller checks. By emulating human behaviour, there should be no difference between humans manually performing actions on a website and Selenium automating those actions.

Sometimes, the human-like behaviour of element interactions is unwanted or the provided actions do not offer enough complexity, which is why the W3C WebDriver API also offers a low-level way of interacting with the web browser called the *Actions API*.[1] The Actions API offers granular actions: pressing or releasing individual keyboard keys, moving the mouse pointer (via mouse or touch-pen motions), pressing or releasing mouse or pen buttons, scrolling using the scroll wheel, and pausing and doing nothing for a set amount of time. A command using the Actions API takes the form of a sequence of actions listed.

[1]Despite being called the Actions API, it is a sub-API of the W3C WebDriver API and not a separate API.

This allows programmers to perform complex actions not possible using element interactions. Clicking and dragging, hovering the mouse over certain elements, and using keyboard shortcuts the website listens for are all examples of actions possible to be performed using only the Action API.

The feature to execute arbitrary JavaScript is not commonly used by end-users. Selenium WebDriver, however, extensively uses it internally to add features not present in the W3C WebDriver API itself, creating an extended "Selenium WebDriver API". The why and how is described in Section 4.1.3.

4.1.2 Browser Drivers

A *browser driver* is a piece of software that implements and provides the W3C WebDriver API described in Section 4.1.1 for a particular web browser. That is to say, browser drivers work as mediators between web browsers and automation code: browser drivers receive W3C WebDriver API commands and translate them into browser-specific actions. Since different web browsers work in wildly different ways, a specific browser driver controls only one type of browser.

In the past, browser drivers were implemented by the Selenium team themselves. But the rising popularity of Selenium WebDriver (in general and among browser developers) combined with the standardisation of the W3C WebBrowser API meant that browser developers themselves started developing and offering browser drivers for their own browsers. Nowadays, with the exception of Internet Explorer, whose driver is still developed by the Selenium team themselves, all browser drivers are supplied by their respective browser developers. Browser drivers may and usually do extend the W3C WebDriver API by adding browser-specific features. These are usually browser-initialisation options, such as starting the browser with some browser extensions enabled.

4.1.3 Selenium Language Bindings and the Selenium WebDriver API

Selenium language bindings are code libraries in a variety of programming languages, such as Java, JavaScript, or Python. These are the code libraries that end-user programmers use to write scripts that automate browser actions. These libraries communicate with browser drivers using the W3C WebDriver API, which in turn control browsers. To the end-user programmer, they expose the *Selenium WebDriver API*, a set of classes and functions for controlling web browsers.

The Selenium WebDriver API is not just a wrapper around the W3C WebDriver API: while the Selenium WebDriver API does indeed provide access to all the functions of W3C WebDriver API, it also offers many functions not provided by it, effectively extending the W3C WebDriver API. For example, the W3C WebDriver API provides many ways of locating elements in the DOM tree. The Selenium WebDriver API in addition provides a way not present in the W3C WebDriver API: relative locators,[2] which enable locating elements in relation to other elements as they appear on the screen. Another feature not present in the W3C WebDriver API is the ability of the Actions sub-API, which is used for low-level browser interaction,[3] to type strings of text as opposed to only being able to press and release individual keyboard keys one by one.[4]

[2] `https://www.selenium.dev/documentation/webdriver/elements/locators/#relative-locators`

[3] see Section 4.1.1 for more information

[4] `https://www.selenium.dev/documentation/webdriver/actions_api/keyboard/#send-keys`

In addition to extending the functionality of the W3C WebDriver API, the Selenium WebDriver API also specifies several convenience functions, for example for comparing and working with colours.[5]

The way how the Selenium language bindings implement the additional features is twofold: either they implement them in the language of the specific language bindings or they implement them as a JavaScript script which gets executed in the browser using the ability of the W3C WebDriver API to execute arbitrary JavaScript code—the JavaScript code is then reused by all implementations of Selenium language bindings. Relative locators are implemented using the JavaScript approach, and the string typing is implemented in the library code by converting the string typing action into a sequence of key presses and key releases.

The reason why the logic of controlling browsers is strewn around so many different places and languages is largely historical and architectural.

Selenium language bindings then ideally implement all the requirements of the Selenium WebDriver API so that all versions of Selenium language bindings offer the same features in the same way, no matter which programming language the end-user programmer chooses. The word "ideally" is important to note: the Selenium WebDriver API is not a rigid, well-defined, and standardised API like the W3C WebDriver API; it is more of an informal description of what Selenium language bindings should provide and how they should achieve it. This means that there exist some differences in what features different versions of Selenium language bindings offer.

In conclusion, Selenium language bindings are the libraries that end-user programmers use to write web-automation scripts. They interface with browser drivers using the W3C WebDriver API (a REST API), but they extend the API with additional features and functions. This extended set of functionality is called the Selenium WebDriver API (a description of classes and functions).

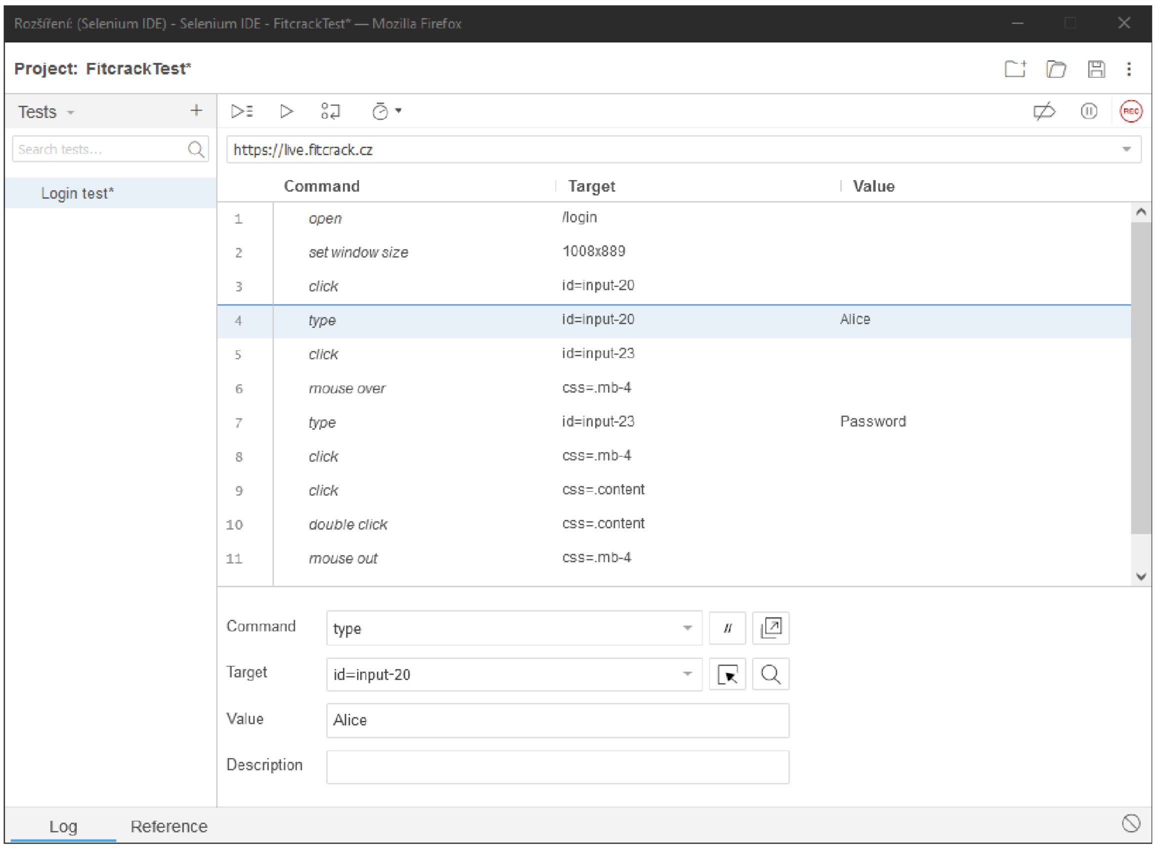
4.2 Selenium IDE

Selenium IDE is a browser extension that allows one to record actions taken on a website to be replayed later. It can record all actions taken by the user, such as clicking a button, hovering a mouse over some element, typing text into form inputs etc. Recordings are referred to as "tests", though they need not necessarily be used for testing purposes. Tests need not be created by recording user actions; they can also be created by manually specifying each individual action. Fig. 4.2 shows the Selenium IDE UI with a recorded test of logging into the Fitcrack Webadmin interface—to be specific, recorded is an attempt to log in using the username "Alice" and password "Password".

In addition to actions, assertions and flow control are also available to be added to the tests. Selenium IDE is not necessary for the playback of tests: saved tests can also be run from a command-line interface using the *Selenium SIDE Runner*, which even allows for running tests in parallel using Selenium Grid. This enables Selenium IDE to automate web-browser actions without the need for scripting in traditional programming languages using Selenium WebDriver.

Though Selenium IDE can be used for creating complex tests and automation code, it is generally recommended to use Selenium IDE for simple tasks, like bug-reproduction scripts. However, Selenium IDE offers the option of exporting tests to equivalent Selenium

[5] `https://www.selenium.dev/documentation/webdriver/support_features/colors/`



Rozšíření: (Selenium IDE) - Selenium IDE - FitcrackTest* — Mozilla Firefox

Project: FitcrackTest*

https://live.fitcrack.cz

	Command	Target	Value
1	*open*	/login	
2	*set window size*	1008x889	
3	*click*	id=input-20	
4	*type*	id=input-20	Alice
5	*click*	id=input-23	
6	*mouse over*	css=.mb-4	
7	*type*	id=input-23	Password
8	*click*	css=.mb-4	
9	*click*	css=.content	
10	*double click*	css=.content	
11	*mouse out*	css=.mb-4	

Command	type
Target	id=input-20
Value	Alice
Description	

Figure 4.2: The main Selenium IDE window showing a recording

WebDriver in a selection of languages, so the exported code can be used as a starting point in building Selenium WebDriver tests.

Chapter 5

Implementation of Page Objects and Browser-Controlling Code

To develop a test suite using the page-object model, the UI of the application under test needs to be modelled as a set of objects that represent each page or major part of a page of the application. As Fitcrack Webadmin is a single-page application, the designed page objects cannot strictly model pages, as the result would be one monolithic object—a quite unhelpful abstraction. Instead, I model each screen of Fitcrack as a page object; some screens are split up into distinct sections, so in those cases, these sections are represented as page objects of their own. Not all features of Fitcrack Webadmin are modelled; only the ones necessary to perform the designed tests are. The page-object model, however, offers extendability, so expanding with additional features should be possible without disruption.

In this chapter, the page objects and other code that directly interacts with the browser is described. In the first section, base classes for page objects are described, then the specific page objects are described, and finally, the code common code used by page objects is described (such as helper functions and custom exceptions).

5.1 Common Page-Object Code

In this section, the base classes from which page objects inherit are described.

5.1.1 Base Class for Page Objects

All page objects defined inherit from the `PageObject` base class. This class provides common features and abstract methods used by all page objects.

One abstract method that is defined is `ensure_loaded`. This method waits until the UI the page object represents is loaded and performing actions in the UI is safe. For example, in Fig. 5.1 we can see an example of Webadmin being in the middle of loading a specific page. Interacting with the UI in this state is impossible as there are no UI elements to interact with, and trying to would cause issues such as raised exceptions or possibly undefined behaviour.

Page objects then perform this wait upon initialisation; this behaviour is implemented in the `__init__` method (Listing 5.1): unless suppressed, whenever a page object is created, its relevant `ensure_loaded` method is run. This makes sure that Webadmin is always in a stable state, ready to be interacted with. If this state cannot be achieved—that is to say

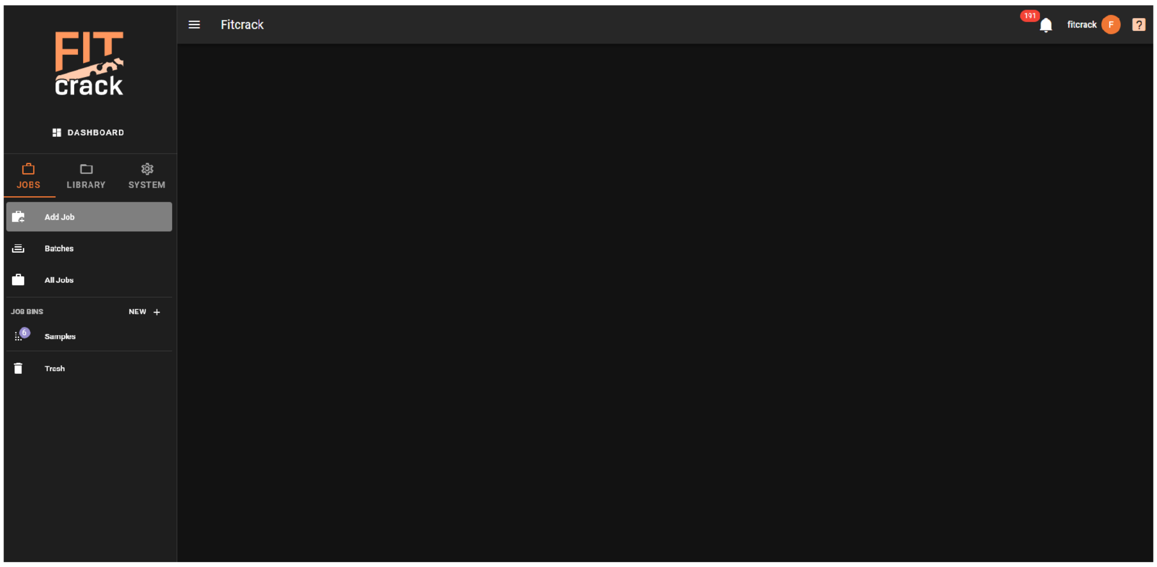


Figure 5.1: A screenshot of Webadmin showing an example of the UI not being fully loaded yet

that the loading of a page of Webadmin fails—the page object raises a relevant exception right upon initialisation.

```python
def __init__(self,driver:WebDriver,no_ensure_loaded=False):
    self.driver = driver
    if not no_ensure_loaded:
        self.ensure_loaded()
```

Listing 5.1: `__init__` method used in the `PageObject` class

The `__init__` method also saves a reference to a WebDriver object, which is required to perform all actions upon the UI, because the web browser has to be controlled using this object.

Then there are provided methods for interacting with parts of Webadmin that are not strictly part of any specific page:

For working with snackbar notifications, the methods `get_snackbar_notification` and `_wait_until_snackbar_notification_disappears` are defined.

It is often useful to inspect the snackbar notification to inspect the result of an action. The `get_snackbar_notification` method returns the text of the notification and the type of the notification, either success (Fig. 5.3) or error (Fig. 5.3). The type is determined by inspecting the applied CSS classes of the notification. It is also possible, by setting the optional parameter `raise_exception_on_error` to `True`, to automatically have the method raise an exception if the displayed notification shows an error. The method waits for up to 10 seconds until a snackbar notification appears; otherwise, the method raises an exception.

Since snackbar notifications can obstruct content behind them (Fig. 5.3), which can cause issues if it is desired to interact with the obstructed content, the method `_wait_until_snackbar_notification_disappears` is provided. The method waits for up to 10 seconds until the snackbar notification automatically disappears; otherwise, a timeout exception

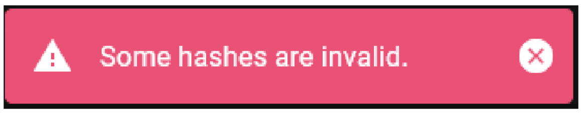
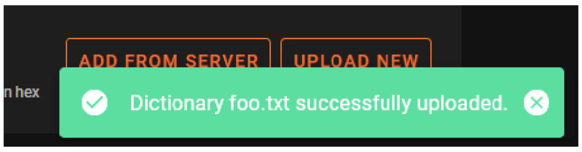

Some hashes are invalid.

Figure 5.2: Example of a snackbar notification showing an error

is raised. Methods in specific page objects use this after performing actions that cause snackbar notifications to appear to hand off Webadmin in a stable state—after performing a method, the UI should be able to be used, so if snackbar notifications are emitted, the methods should wait until they disappear.



Dictionary foo.txt successfully uploaded.

Figure 5.3: Example of a snackbar notification that obstructs two buttons behind it

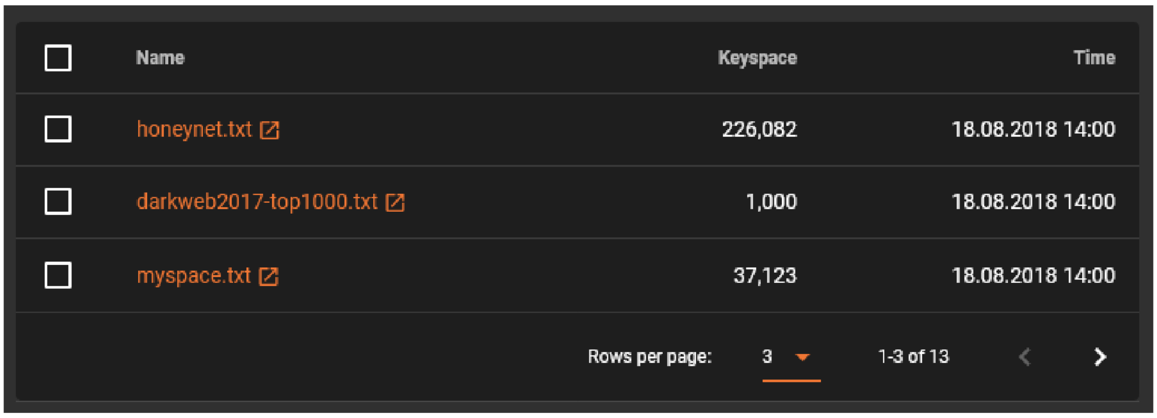
In a similar vein is the last method defined in the base `PageObject` class, `_wait_until_dialog_closes`. This method waits until a dialog box disappears. It waits for up to the default duration of 10 seconds, but the wait time can be set to a custom value. If the dialog does not disappear within the set timeout, an exception is raised. This method is required to be used after exiting out of dialog boxes because exiting plays a dialog-closing animation, and during the animation, the UI cannot be interacted with, so page objects need to wait until the UI is in a stable state. Confirming and closing a dialog box may also initiate lengthy server requests, so the timeout can be adjusted.

5.1.2 Base Class for Page-Component Objects

A subclass of the `PageObject` base class. The `PageComponentObject` class stores an `_element` property with a reference to a DOM element. This element is used as an anchor to locate sub-elements within it. This is useful for when there are many similar components on a page. For example, in Fig. 5.4, there is shown a table, and it is desired to model each table row as an object. By saving a reference to the `<tr>` element of each row into the `_element` property, sub-components can be located more easily. Suppose there is a locator to access the keyspace value in the third column: it can be written as `self._element.find_element(By.CSS_SELECTOR,'td:nth-child(3)')`. Not having an anchor element stored would necessitate the usage of locators that search through the entire DOM tree from the root node.

5.1.3 Table-Row Objects

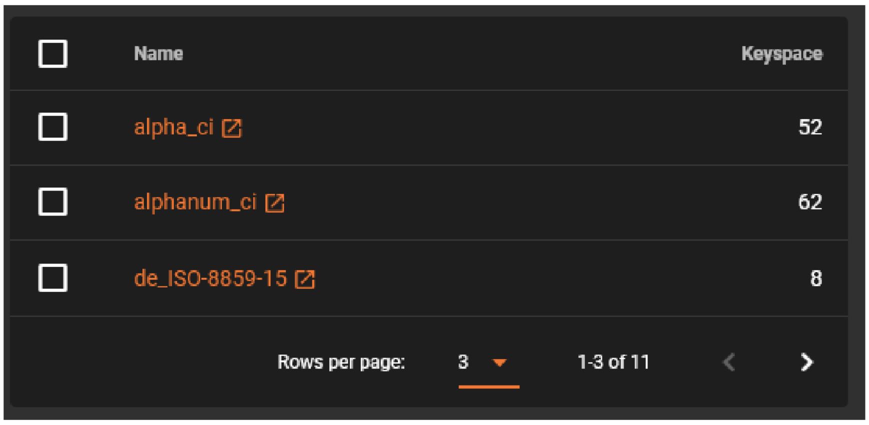
Tables often appear in the Webadmin UI, and interaction with them requires certain special handling. First of, many tables are only slightly different in design and behaviour. See the charset table (Fig. 5.5) and compare it with the table shown in the previous section (Fig. 5.4)). See that while the two tables do contain different columns, their rows all have a check box as the first element. There are many such kinds of tables which share some behaviour but differ slightly, so these families of table are modelled using super objects that model



☐	Name	Keyspace	Time
☐	honeynet.txt	226,082	18.08.2018 14:00
☐	darkweb2017-top1000.txt	1,000	18.08.2018 14:00
☐	myspace.txt	37,123	18.08.2018 14:00

Rows per page: 3 1-3 of 13

Figure 5.4: A table in Webadmin. Each table row can be modelled as a Page-Component Object.



☐	Name	Keyspace
☐	alpha_ci	52
☐	alphanum_ci	62
☐	de_ISO-8859-15	8

Rows per page: 3 1-3 of 11

Figure 5.5: A table in Webadmin showing a selection of charsets.

the common behaviour, and the specific objects inherit from these super objects to model the specific behaviour of the tables.

Tables as a whole are not modelled as objects, however; instead, their rows are modelled. Each row object provides ways to inspect the values of a row's columns and interact with any interactable parts. All table-row objects inherit from `PageComponentObject` (described in the previous section) because each table-row object refers to one specific row and must therefore contain a reference to the specific `<tr>` element.

To create such row objects, there are provided helper functions. Page objects then use these helper functions in their own methods that model user interactions without duplicating table-handling code.

One such helper function is `build_table_row_objects_from_table`. This a higher-order function that takes as input the reference to a DOM object representing a table and a class and iterates through all `<tr>` elements of the table. For each `<tr>` element, it calls the provided class constructor and provides the reference to the `<tr>` element. It then returns a list of objects of the provided class.

However, there are issues with this approach. See the table in Fig. 5.5; notice that three rows are shown, but then notice that there are a total of 11 rows available to be shown. It would be preferable to have access to all of the rows. See also how a table looks if there are no elements to be shown (Fig. 5.6). Using build_table_row_objects_from_table here would not produce the correct results: The table contains a special row that shows that there is no data; this row does not conform to how a regular row with content looks. A table-row object created using a reference to such a <tr> element would cause undefined behaviour. Lastly, a table can be in the state of loading—the table may change its contents while the build_table_row_objects_from_table function is running. This causes the Selenium StaleReferenceException to be raised; the table-row elements the function was working with no longer exist.

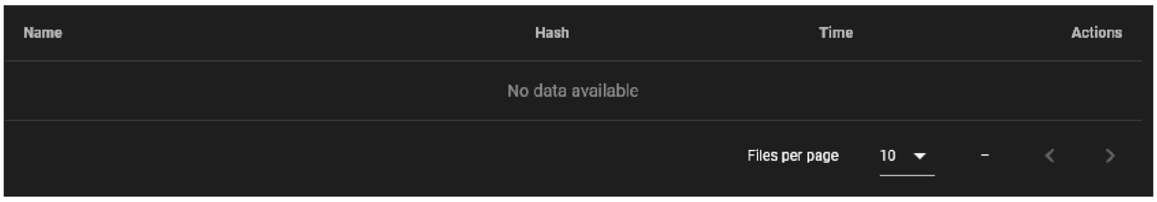


Figure 5.6: A table in Webadmin that is empty; it display no elements.

To provide a robust way to create table-row objects, there is the load_table_elements function. This function first, using the rows-per-page selector sets the table to display all elements (or as many elements as the table can display). After that, it checks whether a special row indicating that the table is empty is present (Fig. 5.6); if yes, it returns an empty list. Then it checks whether the table is in the middle of initial loading; if yes, it raises an exception informing of this UI state. After these checks, the build_table_row_objects_from_table function is called. If a StaleReferenceException is raised, which may indicate that the table is in the middle of loading, then the function retries the steps listed (except selecting the rows per page count) up to 10 times, waiting for 2 seconds between the attempts. If after 10 times a StaleReferenceException is still being raised, then the function raises an exception of its own to indicate that there is a problem and to not be stuck in a loop.

Enableable Table Rows

One family of tables are tables that contain rows that can be enabled with a check box (Figs. 5.4 and 5.5). As such, there is one super class that deals with the common behaviour of all these tables (GenericEnableableTableRow). This class provides one property (enabled), which has a special getter and setter, for interacting with the check box.

Provided is also a helper function called activate_elements_from_table_by_list_lookup. This function takes a list of objects that are compatible with the GenericEnableableTableRow class, a “value getter” function, and a list of values. The function iterates over the list of GenericEnableableTableRow objects and applies the “value getter” function to each; if the value returned by the “value getter” matches a element from the list of values, then that table-row object is set to be enabled (or disabled if the value does not match any element from the provided list). The function then raises an exception if the number of table-row objects that were set to enabled differs from the number of elements in the supplied value list.

Library Table Rows

Another family of tables are the ones for the management of cracking assets as described in Section 2.3. There is thus a super class called `GenericLibraryTableRow`. This class provides two methods for the behaviour that all library table-row objects have in common.

First is a method called `delete` which performs the actions necessary to delete the content associated with the table row. As this action can fail, the method raises a `WebadminError` exception with the text of the shown error message if this happens.

Second is a method called `download`. This method downloads the file associated with the table row. This method takes one Boolean argument (`as_binary`). Depending on the value the argument is set to, the method returns a text or binary string of the downloaded file.

5.2 Specific Page Objects

In this section, the specific page objects used in the test suite are described.

The structure of the code for each page is quite similar. I based it on the Page Factory approach mentioned by the Selenium documentation [12], which is natively supported in the Java version of Selenium.[1] As an example, the code for the `LoginPage` page object is shown (Listing 5.2).

First, any constants used by the page object are defined, if any (Line 2).

Followed are method definitions. The first method that is defined is an implementation of `ensure_loaded`;[2] the method definition may be missing if the page objects does not require require this check (Lines 4–6).

Followed are locators, code that finds elements in the DOM tree (Lines 7–17). These are implemented as methods that return references to DOM elements; the methods are decorated with the Python `@property` decorator. This allows for other methods to access these elements as object properties, which is more natural than having methods named "get element" and having to explicitly call them.

Last are the methods that model the page's behaviour (Lines 19–29). These implement actions the model user may take. All page objects follow this structure.

The sections that follow start, where appropriate, with an abridged (locators removed; only method signatures shown) overview of the structure of the described class as a quick reference of the methods that the page objects supports.

5.2.1 Login Page

The login page (Fig. 2.1) facilities logging in. There are two methods provided: `login` and `navigate`.

The `ensure_loaded` method for this object checks whether the username field can be located.

The `login` method takes two arguments, the username and password. It performs the actions of logging in—it inputs the username and password into the login form and clicks the submit button. Upon successful login, it return two page objects, `SideBar` and

[1]The official Selenium documentation [12] does not offer great detail on the the Java Page Factory support. A better example is included in this community guide (`https://www.browserstack.com/guide/page-object-model-in-selenium#toc4`). The official Selenium documentation itself "[encourages] the reader to search for blogs on these topics".

[2]An abstract method defined in the base class described in Section 5.1.1.

```
class LoginPage(PageObject):
    URL_PATH = '/login'

    def ensure_loaded(self):
        WebDriverWait(self.driver,30,ignored_exceptions={JavascriptException,
            NoSuchElementException}).until(lambda _:self.__username_field)

    @property
    def __username_field(self) -> WebElement:
    return self.driver.find_element(By.CSS_SELECTOR,'input[type="text"]')

    @property
    def __password_field(self) -> WebElement:
        return self.driver.find_element(By.CSS_SELECTOR,'input[type="password"]')

    @property
    def __submit_button(self) -> WebElement:
        return self.driver.find_element(By.CSS_SELECTOR,'button[type="submit"]')

    def navigate(self,prefix:str) -> None:
        self.driver.get(prefix+self.URL_PATH)

    def login(self,username:str,password:str) -> tuple[SideBar,Dashboard]:
        clear_workaround(self.__username_field)
        self.__username_field.send_keys(username)
        clear_workaround(self.__password_field)
        self.__password_field.send_keys(password)

        self.__submit_button.click()
        return SideBar(self.driver), Dashboard(self.driver)
```

Listing 5.2: The `LoginPage` class (documentation strings removed for brevity)

`Dashboard`, as upon logging in, the user is navigated to the dashboard, and the user is always able to interact with the sidebar while logged in. There are no special checks performed to see whether the login attempt was successful or not, but upon unsuccessful login, the `ensure_loaded` methods of the `SideBar` and `Dashboard` objects will time out and raise an exception. Thus a failed login causes failure immediately when calling the `login` method and not later.

The login page is the entry point of interacting with Webadmin, and therefore the `navigate` method is provided. The method opens the login page in the browser, by inputting its URL into the address bar of the browser. The method takes one argument, `prefix`, which is the base URL where the Webadmin instance that the user wants to use for testing is running.

5.2.2 Sidebar

The `Sidebar` page object represents the always-present sidebar in the Webadmin UI (always present after the user is logged in, as can be seen on the left for example in Fig. 2.2).

The `ensure_loaded` method for this object checks whether the Jobs button (at the top, below the Dashboard button) can be located.

The only feature of the sidebar is navigation to other parts of the Webadmin application, so modelled are methods that navigate to those parts and return relevant page objects.

5.2.3 Dashboard

The dashboard (Fig. 2.2) provides an overview of the state of the Fitcrack server. Provided is one method (`get_welcome_text`) that return the welcome text, which informs of who is logged in.

5.2.4 Job-Creation Page

```
class AddJobPage(PageObject):
    def ensure_loaded(self):

    def set_job_name(self,name:str) −> None:

    def get_job_name(self) −> str:

    def create_job(self) −> JobDetailPage:

    def open_input_settings(self) −> InputSettings:

    def open_attack_settings(self) −> AttackSettings:
```

Listing 5.3: Overview of the `AddJobPage` class

The job-creation page (Fig. 2.3) provides the UI to create new tasks. However, many of the features are not modelled in this page object, and that's because the job-creation page is substantial and split into four distinct parts: the input, attack-mode, host-assignment, and additional settings. To better model the UI, these distinct sub-parts are modelled as distinct page objects. This page object thus provides only a few methods: Provided are methods to change and view the name of the job (`set_job_name` and `get_job_name`). And there is provided a method to submit the creation of the job (`create_job`). These features in the UI are not logically part of any of the four distinct sub-parts, so they are provided directly by this object. Provided are also methods that navigate into the four sub-parts of the job-creation dialogue and return their respective page objects (`open_input_settings` and `open_attack_settings`; host mapping and additional settings are not modelled). Descriptions of the page-component objects that model the job-creation page follow.

The `ensure_loaded` method for this object checks whether the job-name field can be located.

Input Settings

The input settings (Fig. 2.4) provide ways to input hashes. Modelled thus are the three different ways to input hashes: inputting a list of hashes (`input_hashes_manually`), inputting a hash-list file (`append_hashes_from_hash_file`), and inputting a file to extract hashes from (`extract_hash_from_file`). Moreover, with the exception of using the feature to extract hashes from files, Fitcrack Webadmin does not perform any automatic hash-type inference. Given that, there also is provided a method for the selection of the input hash

```
class InputSettings(PageObject):
    def select_hash_type_exactly(self,hashtype:str) -> None:

    def get_selected_hash_type(self) -> str:

    def input_hashes_manually(self,hashes:List[str]):

    def get_input_hashes(self) -> List[str]:

    def clear_hash_input(self) -> None:

    def append_hashes_from_hash_file(self,filename:Union[str,Path]) -> None:

    def extract_hash_from_file(self,filename:Union[str,Path]) -> None:
```
Listing 5.4: Overview of the `InputSettings` class

type (`select_hash_type_exactly`). The method takes the name of the hash type; it then interacts with the hash-type selection input, types the hash type into it and then selects a matching hash type from the hash-type selection pop-up. This is necessary because Fitcrack Webadmin requires the hash type to be selected from the pop-up; just directly typing it is not supported.

Provided are also methods that return the state of the UI: the input hashes and also the selected hash type (`get_input_hashes` and `get_selected_hash_type`).

Attack Settings

Next in line are the attack settings (Fig. 2.5). Fitcrack provides different attack modes, and each attack mode has a distinct sub-UI in the grander attack-settings UI. Given that, the different attack-mode sub-UIs are modelled as distinct page objects. This `AttackSettings` object provides only methods that navigate to the different attack-mode sub-UIs and return their respective page objects. These page objects are described in the following sections.

Dictionary Attack

The dictionary attack (Fig. 2.9) has two things that can be set: the selected dictionaries and the selected rule files.

Provided are thus methods for querying the available dictionaries and rule files (`get_available_dictionaries` and `get_available_rule_files`). These methods return table-row objects (as described in Section 5.1.3). Concretely they return `DictionarySe-lection` and `RuleFileSelection` objects, which inherit from the `GenericEnableableTa-bleRow` class; the two classes add methods of accessing the unique columns of the two tables.

So by using these two methods and interacting with the returned table-row objects, it is possible to set the desired dictionaries and rule files. However, for convenience, there are two methods provided (`select_dictionaries` and `select_rule_files`), which allow to provide a list of names of either dictionaries or rule files to be enabled, without having to interact with the table-row objects directly.

The `ensure_loaded` method for this object checks whether the `get_available_dictionaries` method can successfully be executed without raising an exception signifying that the the dictionary table is not loaded yet.

```
class DictionaryAttackSettings(PageObject):
    def ensure_loaded(self):

    def get_available_dictionaries(self) -> List[DictionarySelection]:

    def get_available_rule_files(self) -> List[RuleFileSelection]:

    def rule_file_with_name_exists(self,name:str) -> bool:

    def select_dictionaries(self,wanted_dicts:List[str]) -> None:

    def select_rule_files(self,wanted_rulefiles:List[str]) -> None:
```

Listing 5.5: Overview of the `DictionaryAttackSettings` class

Combination Attack

```
 1  class CombinationAttackSettings(PageObject):
 2      def ensure_loaded(self):
 3
 4      def get_available_left_dictionaries(self) -> List[DictionarySelection]:
 5
 6      def get_available_right_dictionaries(self) -> List[DictionarySelection]:
 7
 8      def select_left_dictionaries(self,wanted_dicts:List[str]) -> None:
 9
10      def select_right_dictionaries(self,wanted_dicts:List[str]) -> None:
11
12      def set_left_mangling_rule(self,mangling_rule:str) -> None:
13
14      def set_right_mangling_rule(self,mangling_rule:str) -> None:
```

Listing 5.6: Overview of the `CombinaionAttackSettings` class

The combination attack (Fig. 2.10) and its representative page object is not dissimilar to the dictionary attack and its page object (Section 5.2.4).

It has similar table access methods (Lines 4–10) for the interacting with the dictionary selections that behave in the same way. Notice that the methods return lists of `DictionarySelection` objects, the same class used by `DictionaryAttackSettings` class; since both of these attacks use the exact same UI for selecting dictionaries, the class can be reused. This is often true for table-row objects.

The only addition are two methods (`set_left_mangling_rule` and `set_right_mangling_rule`) for the setting of the single mangling rules that can be set for both sides of the attack.

The ensure_loaded method for this object checks whether the dictionary-access methods can both be successfully executed without raising an exception signifying that the the dictionary tables are not loaded yet.

Brute-force Attack

The brute-force attack (Fig. 2.11) has the most complicated UI of all of all the attacks and thus has a very complex page-object representation.

The set_mask_value method allows to set one mask's value. The method takes as input the value of the mask to be set and the index of the mask input. Several masks can be input in the UI; index 0 refers to the mask input at the bottom of the mask-input UI, and the mask-input with index $N+1$ is the mask-input directly above mask input N. In a similar vein, the method remove_mask also takes an index and removes the mask with the given index by clicking the corresponding remove button of the mask input with the given index. To add a mask input, the method add_mask_input is provided. This method clicks the "Add Mask" button and thus adds a new mask input.

Given that there can be an unlimited number of mask inputs in the UI, regular locator methods used by page objects cannot be used. For locating mask inputs, there are defined special locators (__get_mask_input_field and __get_mask_remove_button). These locators are not defined traditionally with the @property decorator because these locators take an argument, namely the index of the element they should return. Fortunately, mask inputs and remove buttons all have an HTML ID assigned with the index of their location included in Webadmin, so the locators are simple. And Webadmin also renumbers the IDs upon addition or removal.

For convenience, however, the method set_masks_from_list is provided. This method takes a list of strings representing masks to be input and adds the necessary masks inputs and populates them with the desired values from the provided list. The method also removes any pre-existing masks first.

Then the get_all_input_masks method is provided, which returns a list of strings representing all the input masks.

Webadmin also allows to use stored hashcat mask files as mask input. The get_available_mask_files methods clicks the "Load Masks" button, upon which a dialog box with a table of available mask file appears; the method then returns the names of the mask files that are available. The load_mask_file method takes the name of a mask file and performs the loading process by clicking the "Load Masks", selecting the desired mask file and loading it.

For interacting with the charset and Markov-statistics-file tables, there are provided the methods get_available_markov_files, get_available_charsets, select_markov_file, and select_charsets. These work similar to the table-interaction tables in the dictionary attack settings (Section 5.2.4) with the exception of select_markov_file—as there can only be selected one Markov statistics file (despite the usage of check boxes in the UI), the method does not take a list but a single value.

To interact with the Markov-mode input, two methods are provided (get_selected_markov_mode and select_markov_mode). For the usage with these methods, a special enum class (MarkovMode) is defined and available together with the BruteForceAttackSettings page-object class in the same file. The enum represents the three possbile Markov modes. The get_selected_markov_mode raises an InvalidStateError exception if the

Markov mode cannot be determined (e.g. when no mode is selected or several modes are selected at once, which should not be possible.

Finally, the `set_markov_threshold_value` method sets takes an integer and sets the value of the Markov threshold.

The `ensure_loaded` method for this object checks whether the `get_available_markov_files` and `get_available_charsets` methods can successfully be executed without raising an exception signifying that the the table rows are not loaded yet.

```python
class BruteForceAttackSettings(PageObject):
    def ensure_loaded(self):

    def __get_mask_input_field(self,index:int) -> WebElement:

    def __get_mask_remove_button(self,index:int) -> WebElement:

    def get_available_charsets(self) -> List[CharsetSelection]:

    def get_available_markov_files(self) -> List[MarkovFileSelection]:

    def get_available_mask_files(self) -> List[str]:

    def load_mask_file(self,mask_file_name:str) -> None:

    def select_charsets(self,wanted_charsets:List[str]) -> None:

    def select_markov_file(self,markov_file:str) -> None:

    def get_selected_markov_mode(self) -> MarkovMode:

    def select_markov_mode(self, markov_mode:MarkovMode) -> None:

    def set_markov_threshold_value(self,threshold:int) -> None:

    def set_mask_value(self,mask_value:str,index:int) -> None:

    def add_mask_input(self) -> None:

    def remove_mask(self,index:int) -> None:

    def get_all_input_masks(self) -> List[str]:

    def set_masks_from_list(self,masks:List[str]) -> None:
```

Listing 5.7: Overview of the `BruteForceAttackSettings` class

Hybrid Attack

The hybrid attack (Fig. 2.12) and its page object is quite similar to the combination attack (Section 5.2.4).

For interacting with the dictionary table, there are provided two methods (`get_available_dictionaries` and `select_dictionaries`). These methods work in the selfsame way as the identically named methods in the dictionary attack settings (Section 5.2.4).

```
class HybridAttackSettings(PageObject):
    def ensure_loaded(self) -> None:

    def get_available_dictionaries(self) -> List[DictionarySelection]:

    def select_dictionaries(self,wanted_dicts:List[str]) -> None:

    def set_mangling_rule(self,mangling_rule:str) -> None:

    def set_mask(self,mask:str) -> None:
```

Listing 5.8: Overview of the `HybridAttackSettings` class

Then there are provided methods to set the possible mangling rule and mask for this attack (`set_mangling_rule` and `set_mask`).

Note that there exists only one class representing the hybrid attack settings. The two hybrid attacks with their mirrored UI behave in the same way, and the same locators can be used the same elements in both UIs; this means that a single `HybridAttackSettings` class can represent both attacks.

The `ensure_loaded` method for this object checks whether the `get_available_dictionaries` method can successfully be executed without raising an exception signifying that the the dictionary table is not loaded yet.

PCFG Attack

```
class PCFGAttackSettings(PageObject):
    def ensure_loaded(self) -> None:

    def get_available_pcfgs(self) -> List[PCFGGrammarSelection]:

    def select_pcfg_grammar(self,grammar:str) -> None:

    def get_available_rule_files(self) -> List[RuleFileSelection]:

    def select_rule_file(self,rulefiles:str) -> None:

    def set_keyspace_limit(self,keyspace_limit:int) -> None:
```

Listing 5.9: Overview of the `PCFGAttackSettings` class

The PCFG attack (Fig. 2.13) has several settings that can be configured.

The PCFG and rule-file tables can be interacted with the `get_available_pcfgs`, `get_available_rule_files`, `select_pcfg_grammar`, and `select_rule_file` methods. These work similarly to the table handling methods for the dictionary attack settings (Section 5.2.4 with the exception that the select methods only take one value as input instead of a list values, as the UI can accept only a single value (despite the usage of check boxes).

To set the keyspace limit there is provided the method `set_keyspace_limit`, which takes the keyspace limit as input and sets it.

The `ensure_loaded` method for this object checks whether the `get_available_pcfgs` and `get_available_rule_files` methods can successfully be executed without raising an exception signifying that the the table rows are not loaded yet.

PRINCE Attack

```
1   class PRINCEAttackSettings(PageObject):
2       def ensure_loaded(self) -> None:
3
4       def get_available_dictionaries(self) -> List[DictionarySelection]:
5
6       def select_dictionaries(self,wanted_dicts:List[str]) -> None:
7
8       def get_available_rule_files(self) -> List[RuleFileSelection]:
9
10      def select_rule_file(self,wanted_rulefile:str) -> None:
11
12      def set_minimal_password_length(self,length:int) -> None:
13
14      def set_maximal_password_length(self,length:int) -> None:
15
16      def set_minimal_number_of_elements_per_chain(self,number:int) -> None:
17
18      def set_maximal_number_of_elements_per_chain(self,number:int) -> None:
19
20      def set_keyspace_limit(self, limit:int) -> None:
21
22      def get_password_duplicate_check_state(self) -> bool:
23
24      def set_password_duplicate_check(self,new_state:bool) -> None:
25
26      def get_case_permutation_state(self) -> bool:
27
28      def set_case_permutation_mode(self,new_state:bool) -> None:
29
30      def set_random_rule_count(self,count:int) -> None:
```

Listing 5.10: Overview of the `PRINCEAttackSettings` class

The PRINCE attack (Fig. 2.14) can be configured in multiple ways.

The dictionary and rule-file tables can be interacted with the `get_available_dictionaries`, `get_available_rule_files`, `select_dictionaries`, and `select_rule_file` methods. These work similarly to the table-handling methods for the dictionary attack settings (Section 5.2.4 with the exception that the `select_rule_file` method only takes one value as input instead of a list values, as the UI can accept only a single value (despite the usage of check boxes).

The other methods for setting or checking the state of input values (Lines 12–30) work in a straightforward way in that they take a single value and configure the desired settings accordingly (for setting methods) or simply return the value stored in the UI.

The ensure_loaded method for this object checks whether the get_available_dictionaries and get_available_rule_files methods can successfully be executed without raising an exception signifying that the the table rows are not loaded yet.

5.2.5 Job-Detail Page

```
class JobDetailPage(PageObject):
    def ensure_loaded(self):

    def get_hashes(self) −> List[tuple[str,str]]:

    def start_job(self) −> None:

    def stop_job(self) −> None:

    def get_job_state(self) −> str:

    def get_available_hosts(self) −> List[str]:

    def select_hosts_for_job(self,desired_hosts:List[str]) −> None:

    def get_active_hosts(self) −> List[ActiveHostEntry]:

    def check_if_job_finished(self) −> bool:

    def wait_until_job_finished(self,timeout:float) −> None:

    def get_workunits(self) −> List[WorkunitEntry]:
```

Listing 5.11: Overview of the `JobDetailPage` class

The job-detail page (Fig. 2.8) offers ways to control and inspect the status of a created cracking job.

For control purposes, the methods `start_job` and `stop_job` are provided. These methods, as their name implies are used to start and stop the job by clicking either the start or stop job button.

The state of the job can checked with the `get_job_state` method; the method returns a string containing the state of the job exactly as is shown in the UI. Often, the only reason for checking the state of a job is to see whether a started job ended. For that reason, the method `check_if_job_finished` is provided. The method returns `True` if the job ended and `False` if the job has not ended yet. That is to say that the job state is any of the terminal states (“Finished”, “Exhaused”, or “Timeout”; as described at the end of Section 2.1). Finally, the method `wait_until_job_finished` takes as argument a timeout in seconds and waits for up to that time until the job is ended; if the timeout is reached, a timeout exception is raised.

After the cracking job is done, it is desired to check the results of the job. For this, the `get_hashes` method is defined. This method returns a list of tuples, where the first element is the hash that was being cracked and the second element is the cracked password; if the password could not be recovered, the second element is an empty string.

Managing the assigned hosts of a job is another task that is being modelled. The `get_active_hosts` method inspects the table that shows hosts that are assigned to the job and returns a list of them. It returns of a list of `ActiveHostEntry` objects, which are page-component objects representing rows of the table. Then are provided two methods (`get_available_hosts` and `select_hosts_for_job`). Both interact with the host-assignment dialog box that appears after clicking the "Assign Hosts" button. The former method returns a list of strings of all the names of the hosts that appeared in the dialog box. The latter method takes a list of strings of host names and sets those hosts as active for the job; because this operation may fail, the method checks the snackbar notification that appears after the operation is run and returns a `WebAdminError` exception with the text of the displayed error message.

Finally, the `get_workunits` method returns a list of workunits that Fitcrack created for the cracking job. The method return a list of `WorkunitEntry` objects, page-component objects representing rows of the workunit table.

The `ensure_loaded` method of this page object waits until the `get_job_state` method returns a non-empty string (the method returns the state of the job as shown in the UI, so returning an empty string is an indication that the page has not fully loaded yet).

5.2.6 Library Management

The six cracking-asset library pages in Webadmin are different enough to all be represented by separate page objects. Mainly due to their slightly different locators and table-row objects they interact with. However, the methods and they expose to the end-user-tester are similar in name and identical in function.

```
class RuleFileManagement(PageObject):
    def ensure_loaded(self):

    def get_available_rule_files(self) -> List[RuleFileManagementRow]:

    def upload_rule_file(self,filename:Union[str,Path]) -> None:
```

Listing 5.12: Overview of the `RuleFileManagement` class

Let us look at the structure of the `RuleFileManagement` page object (Listing 5.12, which corresponds to Fig. 2.15). Provided are two methods.

First is `get_available_rule_files`. This method returns the contents of the table shown on the page in the form of a list of table-row objects. Concretely, these are `RuleFileManagementRow` objects, which inherit from the `GenericLibraryTableRow` super class (described at the end of Section 5.1.3). These table-row objects then can be used to perform download and deletion operations with the assets of the library page (these methods are defined in the super class).

Second is the `upload_rule_file` method. This method takes either a Python `pathlib.Path` object or a string representing a file path and uploads the file pointed by that path as a new rule file. This operation can fail; if so, the method raises a `WebadminError` exception with the text of the error shown.

For the management of charsets and mask files, the provided methods are identical, save for the change of method name to accommodate the type of resource files the interact with and the type of table-row object used.

The page object for dictionary management page also provides the same two methods with one difference. The `upload_dictionary` method takes two additional Boolean arguments (`sort_on_upload` and `hex_dictionary`). These correspond to the extra upload options available when uploading dictionaries (Fig. 2.19).

Finally, the page objects for the management pages of the two cracking assets that can be generated from dictionaries (PCFGs and Markov statistics files) provide an additional method (`make_markov_file_from_dictionary` or the respective renaming). This method takes as input the name of a dictionary and generates an asset file from it. The method raises an exception if a dictionary with the given name does not exist. And as this operation can fail, the method raises a `WebadminError` exception with the text of the error shown in that case.

5.3 Shared Code

In this section, the generic code that is used by many page objects is decribed.

5.3.1 The "Click Away" Functions

Sometimes, one needs to "click away" from, say, an input field to return to a neutral application state. For example, when typing into an input field, a pop-up may appear; for example, when using the mask editor (Fig. 5.7). The `click_away` function sends a mouse click command to the browser that clicks on a neutral part of the Webadmin UI that is almost always available to be clicked. This should cause any stuck elements to be removed and the UI to become able to be safely used without fear of UI obstructions.

There also exists the function `click_away_dialog`. When a dialog box is open, the regular `click_away` causes the dialog box to close as it tries to click outside the dialog box. The `click_away_dialog` function works the same way as the regular `click_away` function but uses a target that is always present and neutral in a dialog box to not cause the dialog box to to close.

It should be noted that Selenium methods already perform such "click away" automatically with certain methods like clearing input elements [16, see "unfocusing"]. However, this automatic behaviour sometimes with the modern web elements used by Webadmin and sometimes page objects use lower-level interaction methods that do not perform these automatic steps.

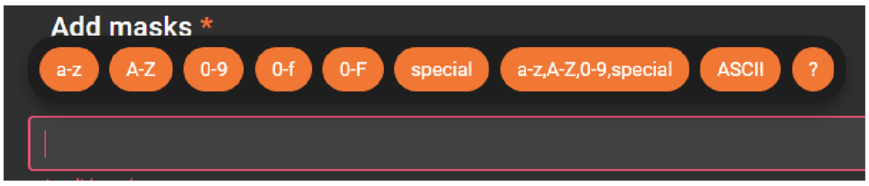


Figure 5.7: The pop-up element of the mask editor obstructs other UI elements

5.3.2 Obstructed Click Workaround

Sometimes in Fitcrack Webadmin, some elements cannot be clicked by using the default Selenium `element.click()` method. This happens when dealing with check boxes and radio buttons: a ripple effect obstructs the actual check box or radio button (Fig. 5.8), causing Selenium to assume that the element would not actually get clicked, throwing an exception.

This is not actually correct and the check box or radio button does get clicked; that is to say that the relevant click events are launched when a mouse cursor click on the location of the "obstructed" element.

To workaround this issue, there exists the function `obstructed_click_workaround`. This function takes a DOM element as input. It first scrolls the element into view and then clicks on it. The function, however, uses the Selenium Actions API to perform these 2 actions—the Actions API does not perform any of the checks of the regular `element.click()` Selenium method. Scrolling an element into view is automatically performed when using the regular `element.click()` method, but this function must perform this action manually.

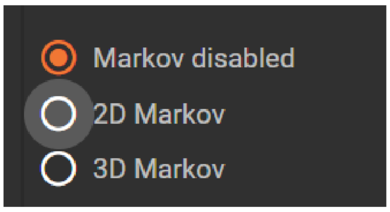


Figure 5.8: The ripple effect around the second radio button obstructs the actual radio-button element.

5.3.3 Querying Check Box State

Check boxes in Webadmin do not have a simple `checked` property that can be used to garner the check box's state as Webadmin uses custom check box elements. The `get_checkbox_state` function takes a Webadmin chec box DOM element as input and returns a Boolean giving the state of the check box and raises an exception if the state cannot be determined. The function determines the state by inspecting the classes applied to the check box.

5.3.4 Input Clear Workaround

For some unknown reason, the default Selenium `element.clear()` method used to clear input elements does not work with inputs in Webadmin. This is a workaround function that works by sending CTRL+A followed by BACKSPACE, thus hopefully selecting everything and then deleting it.

5.3.5 Selenium Scroll Workaround

As of the writing of this thesis, support for the scroll actions using the Actions API is buggy. The Selenium documentation states that the Selenium "scroll to element" method "will need to be used if elements are not already inside the viewport".[3] However, trying to perform this in Firefox does not work—the method raises an exception that the element cannot be scrolled to because it is outside the viewport. This behaviour does not appear in Google Chrome. I reported this apparent bug to the responsible Firefox browser driver developers,[4] and the issue is not yet resolved.

[3]`https://www.selenium.dev/documentation/webdriver/actions_api/wheel/#scroll-to-element`
[4]`https://github.com/mozilla/geckodriver/issues/2078`

Given the current non-portable behaviour of scroll actions in Selenium, I implemented the `scroll_into_view_workaround` function. This takes function takes a DOM element as input and scrolls it into view. It accomplishes by calling the method that takes a screenshot of the element; this method forces a scroll to happen and causes no side effects like other interaction methods that cause scrolls to happen like clicks.

5.3.6 File Download

Selenium does not support downloading files using a web browser. Or more precisely, Selenium can start a download by clicking a download link but does not offer any methods for interacting with the downloaded files. Instead they Selenium documentation recommends to use HTTP request libraries like libcurl for file downloads [11].

The function `download_file_webadmin` is used for this purpose. The function takes as input a download link contained in Webadmin and also the Boolean argument `as_binary` and returns the downloaded file either as text or binary string.

The function uses the third-party `requests` library[5] to perform the download. However, as the user has to be logged into Webadmin to be able to dowload files, the proper authentication data must be provided to perform the file download.

Webadmin uses *Javascript Web Tokens* (JWTs) [5] to authenticate users. To acquire the JWT from the current session, the Selenium method to execute arbitrary JavaScript in browsers is used to run a short script that fetches the value of the JWT. In addition to the JWT, browser cookies also need to be sent to properly authenticate the client to the server. A selenium method is used to extract all cookies from the current session. The acquired JWT and browser cookies are then included with the HTTP request for the file download.

A special consideration has to be made for returning the downloaded files as text strings: When one opens a file in Python, a mode called "universal newlines".[6] This mode converts all types of new lines (Windows, UNIX, and old Mac OS) into a single type—UNIX line endings. This mode is only used for files; HTTP responses converted to text, which is what this function work with, do not perform universal-newline substitution. So the function manually converts the text response to use only UNIX line endings to be consistent between downloaded files and local files.

5.3.7 Near Relative Locator Distance Workaround

As of writing this thesis, the Selenium documentation for the Python implementation of the near relative locator[7] mentions that it is possible to set a custom search distance for use with the near relative locator. However, this is incorrect. The Selenium bindings internally support this feature, but the provided methods do not provide access to this feature.

This function `near_locator_distance_workaround` takes as input a relative-locator object, a web element, and a search distance and by modifying the internal state of the locator object is able to create a replacement for the regular near relative locator function that properly support search distances.

[5]`https://requests.readthedocs.io/en/latest/`

[6]`https://docs.python.org/3/glossary.html?highlight=universal%20newline#term-universal-newlines`

[7]`https://github.com/SeleniumHQ/selenium/blob/c81d86a37b8fecbf1cddf7dd6d047a2b963e1cf6/py/selenium/webdriver/support/relative_locator.py#L134-L138`

5.3.8 Custom Exceptions

There are two custom exceptions defined for page objects to raise if the encounter issues.

The first is `InvalidStateError`. This exception indicates that a page object could not perform an action because of invalid state of the web Webadmin UI. That is to say that the UI is interactable but is in a state that should not be possible. For example, a set of connected radio buttons of which two or more are selected at once.

The other `WebadminError`. This exception indicates that Webadmin showed an error to the user. The content of the error is then placed inside the exception.

Chapter 6

Test Design and Implementation

This chapter describes the design and implementation of the tests in the test suite. First is described the testing framework used to write the tests and the shared code utilised by all tests. Then later in the chapter, each family of tests are described.

6.1 Pytest Testing Framework

These tests are written with the use of the Pytest testing framework [6]. The framework has many features, but described herein are only the features utilised by this test suite.

Pytest test suites can be run using `pytest` command provided by the framework. This command recursively looks through Python files in the current directory and sub directories and finds all Pytest test functions. The test functions then are sequentially run. After all tests have finished, a test report is shown detailing how many tests passed or failed and how they failed.

6.1.1 Test Fixtures

Tests often perform the same behaviour over and over again, chiefly to setup up the environment to be tested, which can be the same among many tests. Pytest offers a way to separate this repeated test behaviour into separate functions called fixtures. Tests can then request those fixtures by specifying their names in the test-function parameters.

In the example (Listing 6.1), the fixture `number` is defined (Lines 1–3); the fixture always returns the number 42. Then there are two test methods defined (Lines 5–9). The function contain use a parameter called `number`. Pytest matches test-function with fixtures of the same name. This means that when the two test functions are run, Pytest calls the test functions with the value of the `number` parameter with the value returned by the `number` fixture.

Fixtures Requesting Other Fixtures

Fixtures themselves can also request other fixtures. In the example (Listing 6.2), there are three fixtures defined. The first fixture is the same as in the previous example. The second and third fixtures request the first fixture and perform some operation on the data provided by it, which they then return as their fixture output. Fixtures are executed only once during the setup for a test. That means that if a test function were to request all

```
@pytest.fixture
def number():
    return 42

def test_multiply_by_zero(number):
    assert (number * 0) == 0

def test_multiply_by_one(number):
    assert (number * 1) == number
```

Listing 6.1: Example of a simple test fixture and tests that use it

three fixtures, the first fixture would be run only once and the returned value cached and provided to the two other fixture and to the test function.

```
@pytest.fixture
def number():
    return 42

@pytest.fixture
def float_number(number):
    return float(number)

@pytest.fixture
def number_as_string(number):
    return str(number)
```

Listing 6.2: Example of test fixtures that request other test fixtures

Clean-Up in Fixtures

Sometimes fixtures need to perform clean-up code after they are run. In the example shown (Listing 6.3), there is defined a fixture that opens a text file and returns a file object. File objects should be closed after use—after a test utilising the fixture is run. To accomplish this, the fixture does not use the `return` keyword, but the `yield` keyword. This means that the fixture is a Python generator that can be resumed. After a test finishes, Pytest resumes all fixtures that are generators. So in our example fixture, the opened file is closed after a test that request the fixture is run.

```
@pytest.fixture
def test_file():
    file = open('foo.txt')
    yield file
    close(file)
```

Listing 6.3: Example of test fixture with clean-up code

6.1.2 Test Parameterisation

Often it is desired to run a test several times but use different values each time. Rather than duplicate the same test several times and then change only the values used in the test, Pytest offers test parameterisation. In the example (Listing 6.4), there is defined a test function which checks that, when a number is multiplied by zero, the result is zero. The test is parameterised using a decorator (Line 1). The first argument to the decorator is a string that tells Pytest which arguments the test function uses should be parameterised. The second argument is a list of values to be used as test parameters. In the test function in the example this means that the test is run three times—the `number` parameter is set to the values 2, 3, and 5 in the three respective test runs.

```
1  @pytest.mark.parametrize('number', [2,3,5])
2  def test_zero_multiply(number):
3          assert (number * 0) == 0
```

Listing 6.4: Example of a parameterised Pytest test

Indirect Parameterisation

In the previous example shown, direct parameterisation was used—the parameterisation data was directly provided to the test method. There also exist indirect parameterisation. This way, the parameterisation data is sent to a test fixture, which then can perform operations on the data and then return a value, which the test uses. In the modification of the previous example (Listing 6.5), indirect parameterisation is used. The parameterisation decorator (Line 5) is provided with the extra parameter `indirect` set to `True` to indicate the usage of indirect parameterisation. The test parameterisation data is then sent to the fixture with the name listed in the first decorator parameter. The fixture (Lines 1–3) then can access the parameterised data through the built-in `request` fixture. In the example, the fixture converts the test parameters it receives to floating-point numbers and returns them. The test function then is run three times with the three converted float values.

```
1  @pytest.fixture
2  def float_number(request):
3      return float(request.param)
4
5  @pytest.mark.parametrize('float_number', [2,3,5], indirect=True)
6  def test_zero_multiply(float_number):
7          assert (float_number * 0) == 0
```

Listing 6.5: Example of a parameterised Pytest test using indirect parameterisation

6.2 The Pytest Selenium Plugin

To use Pytest for the creation of Selenium-based tests, the Pytest-Selenium plugin [1] is used.

The plugin provides a fixture called `selenium`. The fixture returns an initialised Selenium WebDriver object. The plugin also adds a command-line option to Pytest to specify which web browser should be used for the initialisation of the object. So the test suite can be run with whatever web browser is desired.

Another provided fixture is `base_url`. The fixture returns a string representing a URL the tests should navigate to on startup. The plugin adds a command-line option to Pytest, which sets the base URL that is used within the tests. This allows one to run the tests on whichever Fitcrack instance one desires.

6.3 Custom Common Pytest Fixtures

This section describes the custom test fixtures that the tests utilise. Described are common test fixtures in shared usage by many kinds of tests. Fixtures that are used by only a specific set of tests are described later in Section 6.4 alongside the description of their related tests.

6.3.1 Credential Management

All tests start by logging into Fitcrack Webadmin. This requires the input of a username and password. The test suite register a custom pytest command-line option (`-credentials`). A fixture called `credentials` then returns the credentials from the command line or, if the option was not set, returns the default credentials used by Fitcrack.

6.3.2 Navigation to Specific Pages

Several tests interact with one specific part of Fitcrack. For example, tests that test adding cracking jobs interact with the job-creation page and its associated page object (Section 5.2.4). So that tests do not need to repeat code that navigates to the part of Fitcrack they want to interact with, UI navigation code is contained within fixtures. For example, the `add_job_page` fixture navigates to the page and returns the page object representing that page. A test that needs to interact with that page just requests the fixture and does not need to contain the code for navigating to it.

These fixtures utilise the feature of pytest that fixtures can request other fixtures. The `add_job_page` fixture does not contain all the code to navigate to the page. To navigate to the job-creation page, one needs to interact with the sidebar (Section 5.2.2). So the `add_job_page` fixture requests the `sidebar` fixture, and so on. This further reduces code duplication within the navigation fixtures.

Special Consideration for navigating to the Job-Creation Page

Many tests create cracking jobs. If a person logs into the Fitcrack instance that on which tests were run, they will see them in the list of created jobs. To be able to later identify which jobs were created by which tests, the fixture that navigates to the job-creation page also performs another task—it sets the name of the created job to "Job created by an automatic Fitcrack test" followed by the name of the currently running test followed by the current date and time.

6.3.3 Test Files

Tests often interact with test files. Files that are stored on the local computer and are then sent to Webadmin. This brings some problems. Suppose there is a test file called "foo". If a file with the same name already exists on the Fitcrack server, actions like uploading that file can fail. This is can be a problem when re-running the same tests that use the same test file with the same names. If these files still exist for whatever reason after a test is run, re-running the tests to fail. One approach is to delete the test files before or after running a test as clean up. However, this approach does not work because Fitcrack does not actually allow to truly delete resource files. Performing file deletions merely hides the file from the user, but it is still stored on the server and a new file with the same name cannot be uploaded. This odd behaviour (it does not always happen) was discussed with the Fitcrack developer team, and this behaviour is by design.[1]

The solution that this test suite uses is to copy and rename the test file before use. For this purpose there is the `test_file_path` fixture. This fixture takes as input the path (as a Python `pathlib.Path` object to a test file (supplied via indirect parameterisation). The fixture copies the file and appends the current date and time to the file-name stem. It then returns a `pathlib.Path` object to the copied and renamed file. After the test is over, the fixture deletes the copied and removed file to clean-up and not leave needless copies of the same file in the file system.

Finally there two convenience fixtures provided for use with the `test_file_path` fixtures. Those are the `test_file_text_content` and `test_file_binary_content` fixtures. These fixtures return the content of the test file as text or binary string.

6.4 Specific Tests

In this section, the specific tests and their special test fixtures from the test suite are described.

6.4.1 Password Cracking

These tests check the working order of whether cracking jobs can be created, whether can be run, and whether they return valid results in the end. These tests test all the attack modes (Section 2.2). The process, in broad terms, for this test is as follows:

1. Log in to Fitcrack Webadmin
2. Go to the job-creation page
3. In the input settings, manually add a list of hashes and set the hash type
4. In the attack settings, set the parameters of the attack
5. Create the cracking job (host mapping and additional settings are left to the defaults)
6. Check that state of the newly-created job is “Ready”
7. Start the cracking job
8. Wait until the cracking job is finished, or until a predetermined time out
9. If the cracking job is finished, inspect the potentially cracked hashes that their output matches expectation

[1] `https://github.com/nesfit/fitcrack/issues/68`

The Fitcrack developer team already has a set of inputs and expected outputs for testing Fitcrack cracking jobs in the form of a spreadsheet for use with manual testing. This set of end-to-end cracking tests fully automates this part of testing and also allows for easy addition of test inputs—the test inputs are contained within data files, so one can add or remove tests by adding or removing entries in the data files.

Test Fixture and Test Functions

Notice that varying the test input data only largely affects how step 4 of the process is performed. Depending on what kind of attack is to be tested, the configuration process can be completely different, but all the other steps perform the identical actions.

We can thus isolate the common test behaviour into a fixture called `e2e_cracking_test` (Listing 6.6). The fixture requests the `add_job_page` fixture (Line 2) so Steps 1 and 2 are already performed by a generic fixture and the fixture can start on the job-creation page with an `AddJobPage` page object (Section 5.2.4). Using the AddJobPage object, we can navigate into the Input Settings and use the respective page object (Section 5.2.4) to perform Step 3 (Lines 3–5). Then comes Step 4, setting the Attack Settings. This is part of the test that varies with the test input, so the fixture ends its setup process at this point and lets the test function run (Line 7). The test function in this case contains only logic for setting up the attack type (Step 4); it does not perform anything else. Code after the `yield` statement is usually the clean-up code that should be always run, but in this case, the fixture uses it for assertions, so the fixture performs the rest of the test suite only if the test method (which only sets up the Attack Settings) finished successfully without exceptions (Line 9). If the attack is successfully set up, Step 5, the creation of the cracking job, is performed (Line 10). On the same line, the Job Detail Page page object (Section 5.2.5) is returned by the job-creation method. Thereafter Steps 6–9 are performed in a straightforward manner (Lines 11–15).

```
 1 @pytest.fixture
 2 def e2e_cracking_test(add_job_page:AddJobPage,testdata:GenericE2ECrackingTestInput,
       request:_pytest.fixtures.FixtureRequest):
 3     input_settings = add_job_page.open_input_settings()
 4     input_settings.select_hash_type_exactly(testdata.hash_type)
 5     input_settings.input_hashes_manually([x[0] for x in testdata.hashes])
 6
 7     yield
 8
 9     if request.node.rep_setup.passed and request.node.rep_call.passed:
10         job_detail_page = add_job_page.create_job()
11         assert job_detail_page.get_job_state() == ’Ready’
12         job_detail_page.start_job()
13         job_detail_page.wait_until_job_finished(testdata.wait_time)
14         worked_on_hashes = job_detail_page.get_hashes()
15         assert set(worked_on_hashes) == set(testdata.hashes)
```

Listing 6.6: Code listing of the fixture used for end-to-end cracking tests

For each of the attack modes, a test method is defined. As mentioned before, the test function in this case performs only the setup of the Attack Settings. As an example, let us look at the test method for the dictionary attack (Listing 6.7). We can see that the test method uses the `e2e_cracking_test` fixture. The test code only performs the setting up of the attack type given the test data. In a similar vein, there exist test fucntions for every other attack type.

```python
@pytest.mark.parametrize("testdata", testdata)
def test_dictionary(e2e_cracking_test,selenium:WebDriver,add_job_page:AddJobPage,testdata:
     DictionaryTestInput):
    attack_settings = add_job_page.open_attack_settings()
    dictionary_settings = attack_settings.choose_dictionary_mode()

    dictionary_settings.select_dictionaries(testdata.dictionaries)
    dictionary_settings.select_rule_files(testdata.rule_files)
```

Listing 6.7: Code listing of the test code for end-to-end dictionary attacks

Test Data

The test fixture and test functions are parameterised, so they can be run multiple times with different data. For the test data for end-to-end cracking test, there exists a base class (Listing 6.8). This base class is a Python dataclass—this can be thought of as a named tuple as the class only stores properties. The base data class provides the two properties used by the fixture to configure and check the result of the test—first is `hash_type`, a string representing the hash type that should be set; the second is `hashes`, a list of hashes and their expected cracking output in the same format as used by `get_hashes` method of the `JobDetailPage` page object (Section 5.2.5). The last property that is defined in the base class is `wait_time`. This is the predetermined timeout (in seconds) that the fixture uses in performing Step 8. This property need not be set explicitly; it defaults to a value of 10 minutes, which is useable for most tests. If a test input a different time out to perform its cracking job, then the timeout can be custom set for that test input.

```python
@dataclass(frozen=True)
class GenericE2ECrackingTestInput:
    hash_type:str
    hashes:List[tuple[str,str]]
    _: KW_ONLY
    wait_time:float = 600
```

Listing 6.8: Code listing of the base class for the test data of end-to-end cracking tests

Just like there are test functions for every attack mode, there are test data classes for every such mode as well. These dataclasses inherit from the shown base class. As an example, there is shown the test-data class for the dictionary-attack test (Listing 6.9). These contain the properties needed to configure the specific attack mode.

Alongside every test function exist a Python data file that contains a list of such test-data objects. This list is then used as the input for the parameterisation of the test functions, creating a test for each for entry of test data.

```
@dataclass(frozen=True)
class DictionaryTestInput(GenericE2ECrackingTestInput):
    dictionaries:List[str]
    rule_files:List[str]
```

Listing 6.9: Code listing of the test-data class used by the dictionary-attack test

6.4.2 Library Management

This is a family of tests that verify that management of cracking assets works. The basic concept for these tests is that first a new cracking-asset file is uploaded and then after an assertion is performed.

Generic Library Management Tests

As the various library-management pages offer very similar behaviour, many of the tests are also similar. First will be shown an example of tests for the management of charsets (Listing 6.10).

This is an example of tests that upload a valid file that should be accepted by Webadmin and then assertions are performed.

Tests of this type all have a fixture to perform the set-up task (Lines 9–13). This fixture navigates to the relevant asset-library page and uploads a test file. This action is performed before running every test. The fixture uses the `test_file_path` fixture (Section 6.3.3) to always start with a file with a unique name to inhibit filename collisions.

The first test that is a test to assert that the upload did not fail (Lines 15–16). This test does not perform any actual assertion; the only way this test fails is if the set-up fixture encountered a problem, which is what this test checks—if this test fails, there is a problem already with the upload of the file.

The second test (Lines 18–19) asserts that after upload, the newly added asset file appears in the management UI as a new table row.

The third test (Lines 21–25) tries to download the uploaded file back and compares the contents of the test file and downloaded test file to see that they are the same.

The fourth test (Lines 27–31) navigates to a part of the Attack Settings on the job-creation page that contain a table made for interaction with the asset the management of which the method is testing (since the example shows charset management, the test goes to the Brute-Force Attack Settings (Section 2.2.3) to check that the newly uploaded appears in the charset table therein).

The last test (Lines 33–37) tries to delete the newly uploaded asset and asserts that the file no longer appears in the management UI after being deleted.

All these tests are part of a test parameterised test class (Lines 7–8). Parameterising the allows to easily run each test with multiple test files. These tests are run with all supported file extensions for the given asset file (Lines 1–5).

Tested is also the rejection of invalid files (Listing 6.11). At minimum, one and only test is performed—files with the invalid file extensions should be rejected. The test (Lines 5–9) tries to upload a file and checks that a `WebadminError` exception is raised, which indicates that Webadmin showed an error message to the user. The test then checks that the error message is what is being expected, namely the string “This file extension is not allowed”.

```
TEST_FILES = [
    pytest.param(Path('./test/e2e_library/charsets/fc_auto_test_charset_correct.txt'),id='
        file_ext_txt'),
    pytest.param(Path('./test/e2e_library/charsets/fc_auto_test_charset_correct.charset'),id='
        file_ext_charset'),
    pytest.param(Path('./test/e2e_library/charsets/fc_auto_test_charset_correct.hcchr'),id='
        file_ext_hcchr')
]

@pytest.mark.parametrize('test_file_path',TEST_FILES,indirect=True)
class TestCharsetProperUpload:
    @pytest.fixture(autouse=True)
    def charset_management(self, test_file_path:Path, side_bar:SideBar) ->
        CharsetManagement:
        charset_management = side_bar.goto_charset_library()
        charset_management.upload_charset(test_file_path)
        return charset_management

    def test_upload_did_not_fail(self):
        assert True

    def test_charset_appears_in_list(self,test_file_path:Path,charset_management:
        CharsetManagement):
        assert predicate_in_list(lambda x: x.name == test_file_path.stem,
            charset_management.get_available_charset_files())

    def test_download_gives_same_file(self,test_file_path:Path,test_file_text_content:str,
        charset_management:CharsetManagement):
        uploaded_charset = predicate_in_list(lambda x: x.name == test_file_path.stem,
            charset_management.get_available_charset_files())
        downloaded_file = uploaded_charset.download()

        assert test_file_text_content == downloaded_file

    def test_charset_appears_in_attack_settings(self,test_file_path:Path,side_bar:SideBar):
        add_job_page = side_bar.goto_add_job()
        attack_settings = add_job_page.open_attack_settings()
        brute_force_attack_settings = attack_settings.choose_brute_force_mode()
        assert predicate_in_list(lambda x: x.name == test_file_path.stem,
            brute_force_attack_settings.get_available_charsets())

    def test_delete(self,test_file_path:Path,charset_management:CharsetManagement):
        uploaded_charset = predicate_in_list(lambda x: x.name == test_file_path.stem,
            charset_management.get_available_charset_files())
        uploaded_charset.delete()
        with pytest.raises(ValueError):
            predicate_in_list(lambda x: x.name == test_file_path.stem, charset_management.
                get_available_charset_files())
```

Listing 6.10: Code listing of the tests for verifying that charset management works in Webadmin when supplied valid files

The reason that there is only generic test for invalid files is that in most cases, besides the file extension, there are no restriction on what an asset file can contain and be still valid.

```
TEST_FILES = [
    Path('./test/e2e_library/charsets/fc_auto_test_charset_bad_extension.badext')
]

@pytest.mark.parametrize('test_file_path',TEST_FILES, indirect=True)
def test_file_with_bad_extension_should_be_rejected(test_file_path:Path,side_bar:SideBar):
    charset_management = side_bar.goto_charset_library()
    with pytest.raises(WebadminError, match=r"This file extension is not allowed."):
        charset_management.upload_charset(test_file_path)
```

Listing 6.11: Code listing of the tests verifying that invalid charset files are properly rejected

These kinds of tests are part of the test collections for each of the asset types—all library management test collection have similar base tests. What follows are the special tests that are implemented for only specific asset types.

"Make From Dictionary" Tests

Certain types of asset-management pages (PCFGs and Markov statistics files) allow the generation of assets from dictionaries. To test this feature, the same kind of tests shown in Listing 6.10 are used.[2] The difference with these "make from dictionary" tests is that they use a different fixture for the test set-up process(Listing 6.12). With this fixture, first a new dictionary is uploaded through the dictionary-management page (Lines 3–4); the `test_file_path` fixture (Section 6.3.3) is again used to deal with local dictionary file that is being uploaded. Then, the asset file is generated from the dictionary and the page object representing the asset-management page under test is returned (Lines 5–7).

```
1  @pytest.fixture(autouse=True)
2  def pcfg_management(self, test_file_path:Path, side_bar:SideBar) -> PCFGManagement:
3      dictionary_management = side_bar.goto_dictionary_library()
4      dictionary_management.upload_dictionary(test_file_path)
5      pcfg_management = side_bar.goto_pcfg_library()
6      pcfg_management.make_pcfg_from_dictionary(test_file_path.name)
7      return pcfg_management
```

Listing 6.12: Code listing of the set-up fixture for PCFG "make from dictionary" tests

Sorted Dictionary

The dictionary management page offers an option to sort the uploaded dictionary on upload. And there exist tests that verify the functionality of this feature. The collection of tests is nearly identical to the generic collection shown in Listing 6.10. The main distinction is of course that, when uploading the dictionary in the set-up fixture, the sort-on-upload option is enabled.

[2]That is to say with the exception of the test that downloads and compares the generated file to a pre-existing file. This test is not implemented for "make from dictionary" tests

Most test methods are then the same as the generic test methods. There is one distinction, however, and that is the test download-file test. Instead of comparing the downloaded file to the original that was uploaded, the file is compared to a sorted version of the original file.

Loading Newly Created Mask File

In addition to the regular test method for checking that the asset file appears in a table in the Attack Settings, the test collection for mask management offers an additional test method.

Mask files are special in that the way they are used in the UI is different. Most resource files are merely being selected from a table (Fig. 5.4)—setting which resource files are on or off. Mask files, on the other hand, are used at template files in the Brute-Force Attack Settings (Fig. 2.11). Loading a mask file makes the masks appear in the UI. So for that reason, the test collection for mask files in addition that the newly-added mask file appears in the selection table in the Brute-Force Attack Settings also tries to load the mask file in the Brute-Force Attack Settings UI and verifies that the proper masks appear.

More Advanced Invalid Files

In addition to the generic bad-file-extension test (Listing 6.11), there exist more advanced tests for the management of mask files and PCFGs. These two formats have strict requirements on what a valid mask file and PCFG must be. A mask file must be a text file and each line of text must conform to the hashcat mask format;[3] a PCFG file must be be a ZIP file in the correct PCFG format. For these two asset types, special tests exists that try to upload files that have the correct file extension but do not conform to the required format; the tests then assert that Webadmin rejected the files and showed an expected error message.

6.4.3 Hash Input

These tests verify that inputting hashes in the Input Settings (Fig. 2.4) works as intended. Input Settings provide three ways to input hashes. Since the manual input way is used by the end-to-end cracking tests (Section 6.4.1) this test collection assesses the other two input methods.

Hash File Input

For testing input hashes from hash files, there are two tests functions defined. The first is called `test_add_from_file_works`. This function uploads a hash file and then checks that the input hashes match the content of the uploaded file.

The second test function is called `test_add_from_file_works_two_appends`. This test function works similarly to the first, but it uploads two different hash file one after another. The hash-file input works by appending the contents of the hash file to the current input. So after the upload of the two hash files, the test checks that input hashes correspond to the content of the first hash file followed by the content of the second one.

[3]`https://hashcat.net/wiki/doku.php?id=mask_attack#masks`

File Extraction

To test hash-extraction from encrypted files, several test methods are provided (Listing 6.13).

These tests are parameterised. They are run several times with a different encrypted input file; tested is support for every file format supported by Fitcrack. Similar to password-cracking tests (Section 6.4.1), a test-input class us utilised (Lines 1–5). Each test is thus supplied a path to the encrypted file that should be uploaded, the hash that should be extracted from the encrypted file, and the hash type of the extracted hash. Ten test files are used for this collection of tests, all of which are encrypted using the password "password".

All tests start by uploading the encrypted test file and then performing assertions about the resulting output.

The first test (Lines 9–10) simply uploads the encrypted file and performs no assertions. This test thus only fails if the upload of the encrypted file failed.

The second test (Lines 12–15) asserts that the expected hash has been extracted after upload.

The third test (Lines 17–20) asserts that the expected hash type is inferred.

The last test (Lines 22–32) is more in depth. It creates a cracking job with the extracted hash as input. It sets brute-force (Section 2.2.3) as the attack type with the mask `password`—that is to say an attack that tests only candidate password, the literal text "password", which is what all the test file have been encrypted with. It runs the created cracking job and then asserts that the password is recovered.

6.4.4 Host Mapping

This family of tests verifies that Fitcrack correctly assigns hosts to cracking jobs.

The set up process for this family tests is contained within a fixture. The fixture creates a sample cracking job on which then assertions are performed about host-mapping behaviour. After job creation, the Fitcrack navigates users to the Job-Detail Page, where the created job can be controlled, so the fixture returns a page object representing that page (Section 5.2.5) for the tests to interact with. During the test set up, it is also checked that at least 2 hosts are connected to the Fitcrack server–it is not possible to verify that that Fitcrack correctly assign hosts if Fitcrack has one only choice of host to assign.

One group of tests assigns the created job a specific set of hosts and then checks that the Job Detail Page reports that the hosts are assigned to that job. There are several variations of this for all the tested sets of hosts—tested is the selection of no hosts, all hosts that are available, and only one specific host (and in this case, tested is the selection of the first and then the last host that is available to be assigned).

Then there is a test that assigns a single host to the job and starts it. The test then waits for the cracking job to be over. After the job is ended, it inspects the list of workunits that were created during the run of that test and asserts that all of the workunits were assigned to the proper host. This test is parameterised and run twice—once assigning the first available host to the job and then assigning the last available host to the job.

```
@dataclass(frozen=True)
class EncryptedFileTestInput:
    filepath:Path
    expected_hash:str
    expected_hash_type:str

@pytest.mark.parametrize('test_data',TEST_DATA)
class TestEncryptedFile:
    def test_extraction_did_not_fail(self,input_settings:InputSettings, test_data:
        EncryptedFileTestInput):
        input_settings.extract_hash_from_file(test_data.filepath)

    def test_output_has_expected_hash(self,input_settings:InputSettings, test_data:
        EncryptedFileTestInput):
        input_settings.extract_hash_from_file(test_data.filepath)
        extracted_hashes = input_settings.get_input_hashes()
        assert extracted_hashes == [test_data.expected_hash]

    def test_output_has_expected_hash_type(self,input_settings:InputSettings, test_data:
        EncryptedFileTestInput):
        input_settings.extract_hash_from_file(test_data.filepath)
        hash_type = input_settings.get_selected_hash_type()
        assert hash_type == test_data.expected_hash_type

    def test_extracted_hash_should_get_cracked(self,add_job_page:AddJobPage,
        input_settings:InputSettings, test_data:EncryptedFileTestInput):
        input_settings.extract_hash_from_file(test_data.filepath)
        attack_settings = add_job_page.open_attack_settings()
        brute_force_settings = attack_settings.choose_brute_force_mode()
        brute_force_settings.set_masks_from_list(['password'])

        job_detail_page = add_job_page.create_job()
        job_detail_page.start_job()
        job_detail_page.wait_until_job_finished(600)

        assert job_detail_page.get_hashes()[0][1] == 'password'
```
Listing 6.13: Code listing of the tests for the encrypted-hash-extraction feature

Chapter 7

Experimental Assessment of the Test Suite

This chapter describes how the test suite was run on a real deployment of Fitcrack and locally. It assesses the test results and their benefit to the developers of Fitcrack.

7.1 Fitcrack Issues Identified during Test-Suite Development

During the development of the test suite, to make sure that test do actually perform what they are supposed to, the functionality of the tests was tested on a local installation of a development build of Fitcrack.[1] When writing and executing the tests during development, some bugs and issues within Fitcrack were found and reported to the Fitcrack developer team. Some of them were fixed by the developer team and so the fixes were verified when performing experimental assessments of the entire test suite (described in the sections following this one). What follows are the bugs and issues uncovered during test development:

1. During the development of tests for the library management of dictionary files (Section 6.4.2), it was discovered that uploading dictionaries with non-ASCII characters causes Fitcrack to fail with an unhandled-exception error.[2]
2. It was discovered that Fitcrack refuses to upload certain kinds of cracking-asset files with file names that were used in the past.[3] This was confirmed by the Fitcrack team to be expected behaviour though the team changed the error message to make this fact clear. Changes to the design of the tests were required to be made to accommodate the correct behaviour of Fitcrack (Section 6.3.3).
3. The test that uploads a PCFG file and then tries to download it back was failing, discovering that this feature does not work with newly uploaded PCFG files.[4]
4. The test that tries to upload a mask file with invalid content was failing, discovering that verification mask-file format was not working.[5]

[1]Specifically the local installation was running a version of Fitcrack corresponding to commit `f2a0232` in the Fitcrack GitHub repository
[2]`https://github.com/nesfit/fitcrack/issues/66`
[3]`https://github.com/nesfit/fitcrack/issues/68`
[4]`https://github.com/nesfit/fitcrack/issues/73`
[5]`https://github.com/nesfit/fitcrack/issues/74`

5. During the development of tests that try to generate asset files from dictionaries, it was discovered that no checks for duplicate names are performed, corrupting existing asset files with the same name.[6]

6. The tests that try to input hashes by extracting them from encrypted files (Section 6.4.3) were failing with RAR4 files with unencrypted headers[7] and modern password-protected Microsoft Office files.[8] During the development of these tests it was also discovered that Fitcrack Webadmin listed outdated information about the supported files types.[9]

7.2 Running the Test Suite on a Real Deployment of Fitcrack

The test suite was executed on a real development deployment of Fitcrack used by the developers of Fitcrack. The test suite was run on my laptop in parts—not all tests were run at once. It took 3 hours and 45 minutes to run all the tests (155 tests in total, an average of 87 seconds per test). The number of failing tests was 12; the short descriptions of test failures from the test logs are shown in Listing 7.1 (full test logs are contained on the attached storage medium).

What follows is the inspection of all the test failures (the number of the failure corresponds to a line in Listing 7.1, so the first failure is shown on Line 1 in the listing; the second to Line 2 and so on):

- The first test failure is by a library-management test uploading a PCFG file and then downloading it back and comparing that the uploaded and downloaded files are the same (Section 6.4.2). The test failed on grounds that the two files were not identical. However, there is an error with this test and not Fitcrack: the PCFG-file-upload was process was modified to fix a bug described in the previous section, and now the Fitcrack modifies uploaded PCFG files on upload by design.

- Test failures 2–3 of cracking tests (Section 6.4.1) are caused by a missing cracking-resource file. The test inputs provided by the Fitcrack team specify the usage of a cracking-asset file that is not by default included with Fitcrack. When a page-object's method to select the non-existent asset file is used, it raises an exception saying that it could not find any resource file with that name.

- Test failure 4 of a brute-force cracking test happens because Fitcrack incorrectly refuses to create a job with the mask `?1?1?1?1`, showing and error that the mask is in the wrong format. This bug was not present when running tests during development. It was introduced while fixing the bug causing mask files to not be validated described as Item 4 in the list in the previous section.[10]

- Test failures 5–7 of cracking tests are caused by the mismatch between the resulting cracked passwords and the expected cracked password.

[6] `https://github.com/nesfit/fitcrack/issues/76`
[7] Directly disclosed to the Fitcrack developer team during a meeting
[8] `https://github.com/nesfit/fitcrack/issues/81`
[9] Also directly disclosed in meeting
[10] `https://github.com/nesfit/fitcrack/issues/80`

```
 1 FAILED test/e2e_library/pcfg/test_pcfg_file_proper_upload.py::TestPCFGProperUpload::
       test_download_gives_same_file[test_file_path0] - AssertionError: assert b'PK\x03\x04
       \...0\x00\x00\x00' == b'PK\x03\x04\...0\x00\x00\x00'
 2 FAILED test/e2e_cracking/test_brute_force.py::test_brute_force[testdata3] - page_object.
       common.exception.InvalidStateError: Asked to activate 1 elements, but only 0 were found
 3 FAILED test/e2e_cracking/test_brute_force.py::test_brute_force[testdata6] - page_object.
       common.exception.InvalidStateError: Asked to activate 1 elements, but only 0 were found
 4 ERROR test/e2e_cracking/test_brute_force.py::test_brute_force[testdata7] - page_object.
       common.exception.WebadminError: Wrong mask ?1?1?1?1
 5 ERROR test/e2e_cracking/test_pcfg.py::test_pcfg[testdata2] - AssertionError: assert {('2394
       eeac9f...99f8603', '')} == {('2394eeac9f...99f8603', '')}
 6 ERROR test/e2e_cracking/test_pcfg.py::test_pcfg[testdata5] - AssertionError: assert {('2394
       eeac9f...99f8603', '')} == {('2394eeac9f...99f8603', '')}
 7 ERROR test/e2e_cracking/test_prince.py::test_prince[testdata1] - AssertionError: assert
       {('0832bccfc7...8', 'SunSun')} == {('0832bccfc7...8', 'SunSun')}
 8 FAILED test/add_job_page/test_encrypted_file.py::TestEncryptedFile::
       test_extraction_did_not_fail[MS_office_2013] - page_object.common.exception.
       WebadminError: Could not extract hash from file.
 9 FAILED test/add_job_page/test_encrypted_file.py::TestEncryptedFile::
       test_output_has_expected_hash[rar4_encrypted_header] - AssertionError: assert ['
       $RAR3$*0*7d...0cf5b478379a'] == ['$RAR3$*0*1b...0f721c66425c']
10 FAILED test/add_job_page/test_encrypted_file.py::TestEncryptedFile::
       test_output_has_expected_hash[MS_office_2013] - page_object.common.exception.
       WebadminError: Could not extract hash from file.
11 FAILED test/add_job_page/test_encrypted_file.py::TestEncryptedFile::
       test_output_has_expected_hash_type[MS_office_2013] - page_object.common.exception.
       WebadminError: Could not extract hash from file.
12 FAILED test/add_job_page/test_encrypted_file.py::TestEncryptedFile::
       test_extracted_hash_should_get_cracked[MS_office_2013] - page_object.common.
       exception.WebadminError: Could not extract hash from file.
```

Listing 7.1: Excerpt from the collected test logs showing the short descriptions of test failures

- Test failures 8 and 10–12 of encrypted-file input (Section 6.4.3) of modern password-protected Microsoft Office are caused by the fact that Fitcrack cannot extract the hash from the file. The issue was discovered and reported during development (Section 7.1, Item 6), but at the time of running the tests, the issue was not yet fixed, so the test failures are expected.

- Test failure 9 of checking the value of the extracted hash from a RAR4 file with unencrypted header fails because the extracted hash does not match the expected hash. This is however, a problem with the tests as the expected hash was set to the wrong value; the one extracted by Fitcrack is correct.

Running the test suite provided valuable insight for the developers. Bugs in Fitcrack were uncovered. The test run validated that a bug previously reported was fixed (Section 7.1, Item 6); the test run worked also well for regression testing[11], discovering that the bug fix caused a different bug.

[11]Running tests after a change is made to make sure the change does not introduce bugs.

7.3 Re-running the Test Suite after Changes to Fitcrack Were Made

After a while, the Fitcrack developer team fixed some issues discovered in the first test run described in the previous section and updated the development deployment of Fitcrack. The test suite was then run again. This time the test suite was run on my laptop all at once. It took 3 hours and 39 minutes to run (an average of 85 seconds per test). The number of failing tests was now 8 as opposed to the previous 12; the short descriptions of test failures from the test logs are shown in Listing 7.2 (full test logs are contained on the attached storage medium).

```
1 FAILED test/add_job_page_hash_input/encrypted_file/test_encrypted_file.py::
      TestEncryptedFile::test_output_has_expected_hash[rar4_encrypted_header] -
      AssertionError: assert ['$RAR3$*0*7d...0cf5b478379a'] == ['$RAR3$*0*1b...0f721c66425c']
2 FAILED test/e2e_cracking/test_brute_force.py::test_brute_force[testdata3] - page_object.
      common.exception.InvalidStateError: Asked to activate 1 elements, but only 0 were found
3 FAILED test/e2e_cracking/test_brute_force.py::test_brute_force[testdata6] - page_object.
      common.exception.InvalidStateError: Asked to activate 1 elements, but only 0 were found
4 FAILED test/e2e_library/pcfg/test_pcfg_file_proper_upload.py::TestPCFGProperUpload::
      test_download_gives_same_file[test_file_path0] - AssertionError: assert b'PK\x03\x04
      \...0\x00\x00\x00' == b'PK\x03\x04\...0\x00\x00\x00'
5 ERROR test/e2e_cracking/test_pcfg.py::test_pcfg[testdata2] - AssertionError: assert {('2394
      eeac9f...99f8603', '')} == {('2394eeac9f...99f8603', '')}
6 ERROR test/e2e_cracking/test_pcfg.py::test_pcfg[testdata5] - AssertionError: assert {('2394
      eeac9f...99f8603', '')} == {('2394eeac9f...99f8603', '')}
7 ERROR test/e2e_cracking/test_prince.py::test_prince[testdata0] - AssertionError: assert {('22
      fa6121da... 'john'), ...} == {('22fa6121da... 'john'), ...}
8 ERROR test/e2e_library/charsets/test_charset_proper_upload.py::TestCharsetProperUpload::
      test_charset_appears_in_list[file_ext_hcchr] - selenium.common.exceptions.
      TimeoutException: Message:
```

Listing 7.2: Excerpt from the test log showing the short descriptions of test failures

Lines 1–7 correspond to test failures already seen in the previous section. This was expected as the either the fixes for some of the bugs previously discovered were not implemented yet, or those test failures were caused by issues with the tests, not Fitcrack (the test suite was not changed between the two test runs to ensure consistency, but fixes for faulty tests were performed).

Line 8 shows the failure of a test not shown in the previous section. The test failed because of a timeout exception, indicating that some action taken the by the test took too long. Later runs of the same test, however, passed. This indicates that the failure was probably of temporary nature—this is typical in UI testing of complex applications as many factors cause not strictly deterministic behaviour of an application. However, as only one instance of such intermittent failure appeared during the two runs of the test suite, this indicates that the implementation of page objects controlling Fitcrack is rather robust.

The test run provided the Fitcrack developers knowledge that their fixes worked.

Chapter 8

Conclusion

The goal of this thesis was to create a suite of integration tests that test the Fitcrack password-cracking system as a whole. Created were a total of 155[1] tests, testing the major parts of Fitcrack.

Their usefulness has been demonstrated as they discovered about a dozen different bugs and issues in Fitcrack, which the developers of Fitcrack were notified about and are able to fix. The developers were able to verify that their bug fixes worked and they did not cause unintended issues by re-running the test suite.

For the creation of test suite, the page-object model is used. Page objects provide an abstract way for interacting with the Fitcrack Webadmin UI as they provide methods modelling actions the user takes. Tests can use this abstraction of Fitcrack to be rather concise and unchanging even if the UI changes—only the UI-handling code in the page objects needs to be changed.

The work can be further expanded in the future by writing tests for the more minor parts of Fitcrack—given the use of the page-object model described in the thesis, the addition should be easier than starting from scratch as the new tests can reuse page objects already defined.

The test suit needs to be run manually, but further improvements can automate the launch of tests after changes are made to the Fitcrack repository.

The entire test suit takes about three and a half hours to finish. Long test-run times are expected when testing UI, but improvements such as running tests in parallel can be implemented to improve the test-run time.

[1]Including test variations with only different parameters.

Bibliography

[1] Brännlund, J. and pytest-selenium developers. *Pytest selenium documentation* [online]. 2022 [cit. 2023-05-01]. Available at: `https://pytest-selenium.readthedocs.io/en/latest/user_guide.html`.

[2] Fowler, M. *PageObject* [online]. 2013 [cit. 2022-10-22]. Available at: `https://martinfowler.com/bliki/PageObject.html`.

[3] Hranický, R. *Digital Forensics: The Acceleration of Password Cracking*. Brno, Czech Republic, 2022. Ph.D. thesis. Brno University of Technology, Faculty of Information Technology. Available at: `https://www.fit.vut.cz/study/phd-thesis/890/`.

[4] Hranický, R., Zobal, L., Večeřa, V., Múčka, M., Horák, A. et al. *The architecture of Fitcrack distributed password cracking system, version 2*. T-TR-2020-05. Brno University of Technology, Faculty of Information Technology, 2020. 85 p. Available at: `https://www.fit.vut.cz/research/publication/12300`.

[5] Jones, M., Bradley, J. and Sakimura, N. *JSON Web Token (JWT)* [Internet Requests for Comments]. RFC 7519. RFC Editor, May 2015. Available at: `http://www.rfc-editor.org/rfc/rfc7519.txt`.

[6] Krekel, H. and pytest-dev team. *Pytest: helps you write better programs* [online]. 2021 [cit. 2023-04-30]. Available at: `https://docs.pytest.org/en/6.2.x/`.

[7] Leotta, M., Biagiola, M., Ricca, F., Ceccato, M. and Tonella, P. A Family of Experiments to Assess the Impact of Page Object Pattern in Web Test Suite Development. In: O'Conner, L., ed. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. Porto, Portugal: IEEE, October 2020, p. 263–273. DOI: 10.1109/ICST46399.2020.00035. ISBN 978-1-7281-5779-5. Available at: `https://ieeexplore.ieee.org/document/9159053`.

[8] Leotta, M., Clerissi, D., Ricca, F. and Spadaro, C. Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study. In: O'Conner, L., ed. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. Luxembourg, Luxembourg: IEEE, March 2013, p. 108–113. DOI: 10.1109/ICSTW.2013.19. ISBN 978-1-4799-1324-4. Available at: `https://ieeexplore.ieee.org/document/6571616`.

[9] Leotta, M., Clerissi, D., Ricca, F. and Tonella, P. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In: Lämmel, R., Oliveto, R. and Robbes, R., ed. *2013 20th Working Conference on*

Reverse Engineering (WCRE). Koblenz, Germany: IEEE, October 2013, p. 272–281. DOI: 10.1109/WCRE.2013.6671302. ISBN 978-1-4799-2931-3. Available at: `https://ieeexplore.ieee.org/document/6671302`.

[10] Leotta, M., Clerissi, D., Ricca, F. and Tonella, P. Approaches and Tools for Automated End-to-End Web Testing. In: Memon, A., ed. *Advances in Computers*. Elsevier, 2016, vol. 101, chap. 5, p. 193–237. DOI: 10.1016/bs.adcom.2015.11.007. ISSN 0065-2458. Available at: `https://www.sciencedirect.com/science/article/pii/S0065245815000686`.

[11] Software Freedom Conservancy. *File downloads* [online], 16. December 2022 [cit. 2023-04-28]. Available at: `https://www.selenium.dev/documentation/test_practices/discouraged/file_downloads/`.

[12] Software Freedom Conservancy. *Page object models* [online], 9. November 2022 [cit. 2023-04-28]. Available at: `https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/#support-in-webdriver`.

[13] Software Freedom Conservancy. *Selenium History* [online]. 2022 [cit. 2022-11-02]. Available at: `https://www.selenium.dev/history/`.

[14] Steube, J. Inroducing the PRINCE Attack-Mode. In: Mjolsnes, S., ed. *Passwords'14: The 7th International Conference on Passwords*. Trondheim, Norway: Norwegian University of Science and Technology, December 2014, p. 1–42. Available at: `http://passwords14.item.ntnu.no/Preproceedings_Passwords14.pdf`.

[15] Stewart, S. Selenium WebDriver. In: Brown, A. and Wilson, G., ed. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks* [self-published]. Lulu.com, 2011, vol. 1, chap. 16, p. 239–264. ISBN 978-1-257-63801-7. Available at: `http://aosabook.org/en/selenium.html`.

[16] Stewart, S. and Burns, D. *WebDriver*. W3C Recommendation. W3C, June 2018. Available at: `https://www.w3.org/TR/2018/REC-webdriver1-20180605/`.

[17] Weir, M., Aggarwal, S., Medeiros, B. d. and Glodek, B. Password Cracking Using Probabilistic Context-Free Grammars. In: Werner, B., ed. *2009 30th IEEE Symposium on Security and Privacy*. Oakland, Califronia, USA: IEEE, May 2009, p. 391–405. DOI: 10.1109/SP.2009.8. ISBN 978-0-7695-3633-0. Available at: `https://ieeexplore.ieee.org/document/5207658`.

Appendix A

Contents of the Attached Storage Medium

The contents of the attached CD are as follows:

- `/src` — source files of the implemented tests
- `/thesis` — source files for the typesetting of this thesis
- `manual.md` — a manual detailing how to set up and run the implemented tests
- `log1.txt` — the full test log referred to in in Section 7.2
- `log2.txt` — the full test log referred to in in Section 7.3