VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE VÝKONU NÁSTROJE JSHELTER

OPTIMIZING JSHELTER PERFORMANCE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARTIN ZMITKO**
AUTHOR

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

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

147218

Ústav: Ústav informačních systémů (UIFS)
Student: **Zmitko Martin**
Program: Informační technologie
Specializace: Informační technologie
Název: **Optimalizace výkonu nástroje JShelter**
Kategorie: Web
Akademický rok: 2022/23

Zadání:

1. Seznamte se s metodami profilování programů. Zaměřte se na JavaScript a webové aplikace.
2. Seznamte se s architekturou a implementací nástroje JShelter, který funguje jako rozšíření webových prohlížečů.
3. Analyzujte vliv použití nástroje na dobu načítání webových stránek. Za pomocí metod profilování analyzujte kritická místa.
4. Po dohodě s vedoucím navrhněte úpravy nástroje JShelter pro optimalizaci kritických míst.
5. Navržené úpravy implementujte.
6. Experimentálně ověřte přínos vašich úprav.
7. Zhodnoťte dosažené výsledky. Diskutujte další možná vylepšení.

Literatura:

- Nägele, Thomas, et al. "Client-side performance profiling of JavaScript for web applications." *Universidad de Radbound. 2015*.
- Kedlaya, Madhukar N., Behnam Robatmili, and Ben Hardekopf. "Server-side type profiling for optimizing client-side JavaScript engines." *ACM SIGPLAN Notices* 51.2. s. 140-153. 2015.
- Selakovic, Marija, and Michael Pradel. "Performance issues and optimizations in javascript: an empirical study." *Proceedings of the 38th International Conference on Software Engineering*. 2016.

Při obhajobě semestrální části projektu je požadováno:
Body 1 až 4

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hranický Radek, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 10.5.2023
Datum schválení: 31.10.2022

Abstrakt

Cílem této práce je zmírnit dopad nástroje JShelter na výkon při prohlížení webu. Toho bylo docíleno měřením výkonu nástroje JShelter a analýzou jeho kritických míst s největším vlivem na plynulost prohlížení, na jejímž základu byly navrhnuty a implementovány optimalizace, které plynulost zlepšily se zachováním stejně vysoké úrovně ochrany. Mechanizmy ochran založené na zpracování objemných zvukových a obrazových dat byly zrychleny rozšířením o efektivnější implementaci využívající technologii WebAssembly, což přineslo až padesátinásobné zrychlení. Celkový výkon načítání stránek byl zlepšen úpravami mechanizmů načítání konfigurace a zavádění ochrany do stránek, průměrně o 13,5 %.

Abstract

This thesis aims to lessen the impact of JShelter on the browsing experience. The goal was accomplished by measuring the performance of JShelter and analyzing the hotspots that impacted browsing performance the most. Finally, optimizations based on analysis results while leaving the high provided level of protection uncompromised were proposed and implemented. JShelter's fingerprinting protections based on image and audio data processing were optimized by extending them with a more efficient implementation using WebAssembly, which, in some cases, was faster by up to 50 times. The performance while loading web pages was increased with changes in configuration loading and injection mechanisms, on average by 13.5 %.

Klíčová slova

JShelter, fingerprinting, rozšíření prohlížeče, webová bezpečnost, JavaScript, profilování, optimalizace, WebAssembly, AssemblyScript

Keywords

JShelter, fingerprinting, browser extension, web security, JavaScript, profiling, optimization, WebAssembly, AssemblyScript

Citace

ZMITKO, Martin. *Optimalizace výkonu nástroje JShelter*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Hranický, Ph.D.

Optimalizace výkonu nástroje JShelter

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Hranického. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Martin Zmitko
10. května 2023

Poděkování

Chtěl bych poděkovat panu doktoru Hranickému za vedení této práce, za poskytnutí cenných rad a připomínek při řešení, a za pořádnou výzvu ve formě tohoto zadání, která mi umožnila rozšířit své obzory v oblastech, kde bych to nikdy nečekal.

Obsah

Seznam obrázků

Kapitola 1

Úvod

V dnešní době je pro většinu lidí používání internetu téměř nutností. Webový prohlížeč je nepostradatelný nástroj, pomocí kterého jsme ve spojení s okolním světem. Narůstající uživatelská základna ale přitahuje i nekalé aktéry, kteří se kvůli vlastnímu obohacení nebojí ohýbat pravidla a vystavují uživatele zbytečnému nebezpečí. Jedním ze způsobů narůstajících na popularitě je neoprávněné sledování uživatelů. Webové prohlížeče jsou dnes již komplexní kusy softwaru, které obsahují mnohé rozhraní pro komunikaci s webovými stránkami a jsou jim schopny poskytovat širokou škálu informací pro zlepšení uživatelského zážitku. Tyto informace se ale dají zneužít k vytvoření tzv. otisku (angl. *fingerprint*, proces jeho tvorby se nazývá *fingerprinting*) uživatele a jeho prohlížeče. Vzhledem ke komplexnosti poskytovaných informací a neustále se vyvíjejícím technikám tvoření otisků je možné vytvořit s dostatečnou přesností natolik unikátní otisk, že s jeho využitím lze vytvořit nejen přesnou historii prohlížení, ale i komplexní obraz o uživateli jako takovém. Takový obraz může být pro mnoho aktérů (např. pro reklamní společnosti, totalitní vlády nebo hackery) nedocenitelný.

V této práci se věnuji optimalizaci nástroje JShelter, rozšíření prohlížečů pro komplexní ochranu proti fingerprintingu a jiným hrozbám na webu. Toho je docíleno pomocí několika nastavitelných způsobů, zahrnujících omezení poskytovaných informaci, jejich znepřesnění, případně úplného zfalšování, blokování neoprávněných síťových přístupů nebo informování uživatele o pokusech o fingerprinting. Taková úroveň ochrany je ale podmíněna vysokými výpočetními požadavky, což může v některých případech výrazně zpomalit načítání a prohlížení stránek.

Proto jsem se seznámil s nástrojem JShelter a jeho mechanizmy ochrany, analyzoval ho s využitím metod pro profilování a na základě výsledků analýzy jsem navrhl a implementoval optimalizace nalezených kritických míst se zachováním poskytované vysoké úrovně ochrany. Funkce vnášející nepřesnosti do obrazových a zvukových dat tvořených některými zranitelnými rozhraními byly optimalizovány s využitím technologie WebAssembly, která umožňuje takové zpracování dat provádět velmi rychle. Pro zajištění ochrany byla stávající implementace ponechána a byla rozšířena o novou v jazyce AssemblyScript, efektivnější, ale se stejnými výsledky. Zrychlení bylo v tomto případě až padesátinásobné. Načítání stránek bylo zrychleno pomocí úprav mechanizmů pro získání konfigurace pro stránku, vkládání vlastního kódu a zavedení ochrany. Tyto optimalizace byly efektivní, průměrný výkon načítání vybraných měřených stránek vzrostl o 13,5 %.

Práce je strukturována následovně: kapitola 2 se zabývá metodami měření výkonu počítačových programů, zejména jazyka JavaScript. V kapitole 3 rozvádím principy fungování nástroje JShelter a hrozby, proti kterým chrání. V kapitole 4 popisuji postup analýzy kri-

tických míst nástroje JShelter s použitím nástrojů pro vývojáře v prohlížeči a pomocí ruční instrumentace důležitých volání. Na základě výsledků analýzy v kapitole 5 navrhuji optimalizace pro zrychlení prohlížení při užívání rozšíření využitím technologie WebAssembly pro náročné zpracování dat a úpravami mechanizmů načítání konfigurace a vkládání vlastního kódu na stránky. Návrh využívám v kapitole 6, kde popisuji vlastní implementaci. V kapitole 7 se věnuji měření a zhodnocení provedených změn.

Kapitola 2

Měření výkonu programů

V této kapitole se budu věnovat principům a možnostem měření výkonu programů, neboli profilování. Profilování je metoda dynamické analýzy kódu, kdy je monitorován běh programu a jsou sbírány informace jako doba běhu, využití paměti, průběh vykonávání programu nebo typické použité instrukce při běhu. Výstup profilování se nazývá profil programu. Tyto informace jsou často při vývoji kritické, protože poskytují vývojáři důležitý vhled do vnitřního fungování programu. Informace může dál využít např. k hledání chyb, ale především k optimalizaci programu. Díky profilu vývojář nalezne kritická místa spotřebovávající neúměrné množství zdrojů počítače jako procesorový čas nebo paměti. Na tyto místa se může dále zaměřit a zlepšit výkon svého programu [6, 17].

2.1 Metody profilování

Pro účely profilování se používají různé metody. Každá z těchto metod se na profilování dívá z odlišného úhlu pohledu a má svá pozitiva a negativa. Pro různě technologie může být možnost použití metod omezená, metody se ale dají také kombinovat, takové nástroje pro profilování se nazývají hybridní [14].

2.1.1 Profilování založené na sledování událostí

Tato metoda poskytuje informace o běhu programu na základě sledování předem definovaných událostí. Tyto události jsou např. volání, návrat, načtení a opuštění třídy a výjimka. Vzhledem k povaze této metody, což je sledování událostí na vyšší úrovni abstrakce, je využitelná hlavně u interpretovaných jazyků. Při běhu programu poskytují spuštěné události informace interpretu, který je dále zpracovává. Výhodami jsou přesnost poskytovaných informací a minimální vliv na výkon programu, tato funkcionalita ale musí být vestavěna do interpretu profilovaného jazyka. Příkladem je profilování jazyka Java s využitím rozhraní pro profilování virtuálního stroje Java (angl. *Java Virtual Machine Profiler Interface*, JVMPI) [13].

2.1.2 Instrumentace

Instrumentace je metoda profilování založená na vkládání instrukcí do zdroje programu, které při jejich provedení tuto informaci oznamují spuštěnému nástroji pro profilování. Instrumentace může být provedena různými způsoby:

- **Binární instrumentace** – proces vkládání strojových instrukcí do již přeloženého spustitelného souboru.
- **Instrumentace provedená překladačem** – instrukce jsou vloženy překladačem při kompilaci programu.
- **Middleware instrumentace** – program zůstává nezměněný, ale sdílené knihovny jsou nahrazeny knihovnami se stejným rozhraním obohacenými o kód instrumentace.
- **Instrumentace zdrojového kódu** – kód pro instrumentaci je vložen přímo do zdrojového kódu programu, a to buď manuálně vývojářem, nebo nástrojem pro takovou instrumentaci.

Velkou výhodou této metody profilování je vysoká přesnost poskytovaných dat, možný vysoký rozsah sbíraných informací a použitelnost v takřka všech technologiích a jazycích. Nevýhodou je velký dopad na rychlost provádění programu, což může vést k odlišnému chování instrumentovaného kódu [24].

2.1.3 Vzorkování

Tato metoda používá k profilování periodické sbírání vzorků o běhu programu. Vzorky obsahují obyčejně adresu právě vykonávané instrukce a stav zásobníku volání, mohou ale také zahrnovat využití paměti či jiné dodatečné informace. Z těchto dat poté nástroj pro profilování vytvoří statistickou aproximaci průběhu programu. Nevýhoda této metody je velká potenciální nepřesnost výstupu. Z podstaty vzorkování vyplývá, že změny v průběhu programu provedené mezi vzorky nejsou zachyceny. Důsledkem jsou nepřesnosti ve změřené délce vykonávání funkcí a jejich počtu volání. Funkce, které mají nízkou časovou komplexitu a vykonávají se rychleji, než je vzorkovací frekvence, se nemusí ve výsledcích profilování objevit. Tomu se dá částečně zamezit volbou rychlejší vzorkovací frekvence, takové zvýšení ale také zahrnuje zvýšení výpočetní a paměťové náročnosti profilování. Vzorkování vůbec nezasahuje do průběhu vykonávání programu a je použitelné na širokém okruhu jazyků a technologií, což jsou jeho výhody [23].

2.2 Profilování jazyka JavaScript

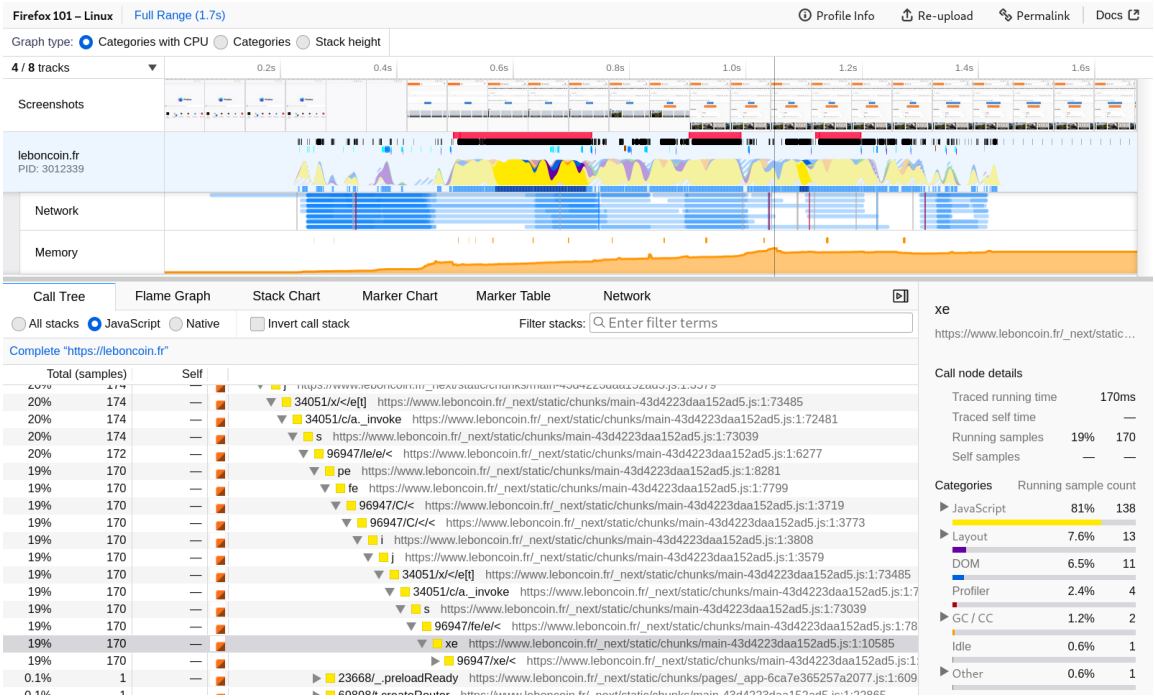
JavaScript je dynamický, objektově-orientovaný, slabě typovaný skriptovací jazyk s masivním využitím ve webových technologiích. Podle statistik ho používá kolem 98 % webových stránek[1], kde se stará o dynamický obsah. Jeho využití stoupá i na serverech a v jiných klientských aplikacích mimo prohlížeč s nástupem technologie Node.js[2]. JavaScript používá Just-In-Time kompilaci, což je způsob spouštění, který kombinuje kompilaci předem a interpretaci. Před spuštěním je kód přeložen do mezikódu zvaného *bajtkód* (jazyk blízký strojovému kódu, přenositelný mezi platformami), který je dále interpretován pomocí izolovaného virtuálního stroje (v kontextu JavaScriptu anglicky nazývaném *engine*) [4].

Protože JavaScript běží ve speciálních virtuálních strojích, tak tyto stroje musí obsahovat potřebné nástroje, aby bylo možné jeho běh profilovat [4]. JShelter je rozšíření do prohlížeče, tedy jediná možnost jeho profilování je v prohlížeči, resp. v jeho virtuálním stroji

[1] https://w3techs.com/technologies/history_overview/client_side_language/all
[2] https://nodejs.org/

pro běh JavaScriptu. Stroje v podporovaných prohlížečích rozšířením jsou V8[3] v prohlížečích založených na jádře Chromium a Spidermonkey[4] v Mozilla Firefox. V8 nástroje pro profilování přímo integruje, součástí Spidermonkey nejsou, ale jsou dostupné jako oddělený komponent prohlížeče Firefox s názvem Gecko Profiler (což mimo jiné umožňuje profilování nativních volání prohlížeče a to i v kombinaci s JavaScript kódem stránek). Tyto nástroje používají metodu vzorkování (viz část 2.1.3).

Jako nadstavba nad těmito nástroji fungují nástroje pro vývojáře implementované prohlížeči[5][6]. Tyto nástroje mimo grafického uživatelského rozhraní pro profilování implementují také jiné pomůcky užitečné vývojářům, jako prohlížení zdrojových kódu, ladění, JavaScript konzole, nahrávání a prohlížení síťových požadavků aj. Profilování v nástrojích pro vývojáře v prohlížečích je cíleno hlavně na vývojáře webových stránek, tudíž se zaměřuje zejména na rychlost načítání a funkčnosti webových komponentů a ladění JavaScriptu, který je součástí stránek. Ukázka výstupu profilování v prohlížeči Firefox je na obrázku 2.1.



Obrázek 2.1: Ukázka výstupu profilování v prohlížeči Firefox.[7]

Další možností profilování JavaScriptu je ruční instrumentace kritických míst. K tomu je využitelné rozhraní vysoce přesného času [7], zejména funkce `performance.now()`. Tato funkce vrátí objekt `DOMHighResTimeStamp`, který reprezentuje čas v milisekundách (s přesností ještě vyšší, definovanou implementací) uplynutý od vytvoření kontextu spuštění. Takto změřený čas je nezávislý na času počítače. Pomocí této funkce jde změřit doba vykonávání kritických míst s větší přesností než při profilování vzorkováním a s garancí správného výsledku.

[3] https://v8.dev/docs
[4] https://firefox-source-docs.mozilla.org/js/index.html
[5] https://developer.chrome.com/docs/devtools/
[6] https://firefox-source-docs.mozilla.org/devtools-user/
[7] Převzato pod licencí MPL-2.0 z https://github.com/firefox-devtools/profiler

Kapitola 3

JShelter

JShelter [19, 20] je rozšíření do webových prohlížečů Mozilla Firefox, Google Chrome a ostatních založených na jádře Chromium s cílem spolehlivě ochránit uživatele před útoky na jejich soukromí a informovat je o provedených pokusech. Rozšíření je napsané v jazyce JavaScript a je zdarma dostupné na tržištích kompatibilních prohlížečů[1][2][3]. Část 3.1 popisuje potenciální hrozby, proti kterým JShelter bojuje. Jednotlivým modulům poskytujícím ochranu proti těmto hrozbám se věnuje část 3.2.

3.1 Hrozby ohrožující soukromí uživatele

S postupem času je zkoumán a vyvíjen neustále se zvyšující počet různých vektorů útoku na uživatelské soukromí na internetu. To vede k narůstajícímu potenciálu odhalení citlivých dat a sledování, ať už necílenému k odhalení zájmů a chování široké skupiny lidí na internetu pro marketingové účely, nebo cílenému k vypátrání individuálního uživatele ze strany hackerů, nebo vládních agentur [12]. Následuje popis čtyř hrozeb, kterým se JShelter snaží zabránit.

Hrozba 1: sledování uživatelů

I přes snahy vlád (zákony pro ochranu soukromí, GDPR) nebo technologických společností (zakázání použití cookies třetích stran) je hrozba sledování stále velmi aktuální. Sledování na základě souborů cookies se nazývá stavové – cookies jsou uloženy na straně uživatele a webové stránky k nim mají přímý přístup [15]. Jako reakce na zákaz cookies pro sledování se urychlil vývoj nové bezstavové metody – tvorba otisků zařízení a prohlížeče neboli fingerprinting [12]. Ten umožňuje velmi přesně identifikovat neochráněného uživatele a sledovat ho napříč internetem. Dále se fingerprinting dělí na dva přístupy:

Pasivní přístup využívá k identifikaci statické informace o uživateli, jeho prohlížeči a zařízení, posílané v HTTP hlavičkách a jiných síťových požadavcích, které by byly posílány nezávisle na pokusech o fingerprinting, nebo poskytované různými rozhraními (API) prohlížeče. Takové informace mohou zahrnovat např. řetězec agenta uživatele, rozlišení obrazovky, jazyk, časovou zónu, systémové fonty, parametry hardwarové akcelerace aj.

Aktivní přístup využívá vlastních JavaScript modulů na stránkách a k tvorbě otisku používá algoritmické metody. Ty jsou schopné identifikovat uživatele na základě funkcionality poskytované specifickými API prohlížeče. Mezi tyto rozhraní se řadí např. Canvas, WebGL,

[1] https://chrome.google.com/webstore/detail/jshelter/ammoloihpcbognfddfjcljgembpibcmb
[2] https://addons.mozilla.org/firefox/addon/javascript-restrictor/
[3] https://addons.opera.com/extensions/details/javascript-restrictor/

WebAudio nebo Sensor. Aktivní metoda funguje nejčastěji na základě identifikace malých rozdílů v obrazových či zvukových datech generovaných specifickým způsobem, zanesených kvůli rozdílům mezi použitými hardwarovými komponenty v jednotlivých zařízeních.

Hrozba 2: velmi bohaté rozhraní prohlížečů

Prohlížeče jsou schopny poskytovat širokou škálu funkcionality a informací o sobě a zařízení, na kterých běží, blížící se nativním aplikacím. Toto umožňuje webům rozšířit svou funkcionalitu čtením hodnot fyzických senzorů nebo interakcí s periferiemi jako je kamera, mikrofon, gamepad a headset pro virtuální realitu. Taková volnost ale otevírá prostor tato rozhraní zneužít pro přímé sledování (přesné polohy, odposlouchávání mikrofonu nebo na základě dat ze senzorů) a tvorbu otisků pomocí zjištění připojených periferií [20].

Hrozba 3: skenování lokální sítě

Ač jsou zařízení v lokální sítí běžně skryté před zbytkem internetu pomocí NAT, za určitých okolností lze využít prohlížeče jako prostředníka k jejich odhalení. I přes politiku stejného původu implementovanou v prohlížečích, která by měla takovým požadavkům zabránit, byly úspěšně provedeny útoky vedoucí až ke vzdálenému spuštění kódu na lokálním síťovém zařízení ze zařízení na internetu [2].

Hrozba 4: mikroarchitekturální útoky

Mikroarchitekturální útoky obecně umožňují odhalení nedávných akcí provedených na zařízení. Toho docílí pomocí specifických metod práce s pamětí a přesného časování provedených akcí. Počet takových metod a proveditelných útoků je velký, následující seznam obsahuje výpis vybraných úspěšné provedených útoků:

- odhalení zobrazeného obsahu pomocí časování odpovědí od serveru nebo doby vykreslování [3],
- přístup k paměti mimo izolovaný prostor alokovaný pro běh stránky s pomocí optimalizací provedených operačním systémem a časováním přístupů [8],
- odhalení historie prohlížení na základě časování vykreslování prvků s navštívenými odkazy [22].

3.2 Moduly nástroje JShelter

JShelter je rozdělený na několik modulů, každý se specifickou funkcí pro ochranu uživatele. Kromě těchto modulů také obsahuje uživatelské rozhraní pro veškerou konfiguraci a kód běžící na pozadí, zajišťující správné nastavení jednotlivých modulů a úrovní ochrany, uživatelsky nastavitelných na základě jednotlivých domén i stránek.

3.2.1 JavaScript Shield

JavaScript Shield (JSS) [20] je hlavní modul nástroje JShelter a poskytuje ochranu proti hrozbám 1, 2 a 4. Fungování tohoto modulu je založeno na modifikaci zranitelných koncových bodů celé řady rozhraní v prohlížečích. Toho je docíleno obalením některých nativních

objektů a funkcí vlastním kódem, který upravuje jejich sémantiku, např. vrací falešné hodnoty, navrácená data mírně upraví, nebo obalenou funkcionalitu úplně deaktivuje. Počet obalených koncových bodů rozhraní je momentálně 113.

Ochrana spočívající v úpravě vrácených dat (použitá v rozhraních Canvas, WebGL a WebAudio, zapadající do aktivního fingerprintingu v hrozbě 1) se dá rozdělit do dvou kategorií. První kategorie (do ní spadají ochrany použité na úrovni Recommended) zanáší do návratových hodnot malé lži, ty ale nejsou detekovatelné uživatelem a pro algoritmy zajišťující fingerprinting jsou detekovatelné pouze s obtížemi. Tyto lži jsou tvořeny na základě výstupu pseudonáhodného generátoru čísel. Jako počáteční hodnota je nastaven hash kombinace jména domény, identifikátoru sezení a cyklického redundantního součtu původních dat, což zapříčiní vždy stejný výsledek v průběhu celého sezení při modifikaci stejných dat na stejné doméně. To jednak dělá uživatele méně identifikovatelným ztížením detekce pokusů o ochranu proti fingerprintingu (výsledek opakovaného volání funkce pro vrácení stejných dat by měl zůstat stejný), také to ale znemožní útočníkovi získat otisk opakovaným voláním a rekonstrukcí původních dat, kterou by byl schopen v případě úplně náhodných modifikací vytvořit.

Druhou kategorií (ochrany na úrovních Strict a Experimental) je generování úplně falešných dat, generovaných taktéž pseudonáhodně se stejnou počáteční hodnotou. V tomto případě jsou data ale znehodnocena a omezí se funkčnost mnoha webových stránek.

Pasivnímu fingerprintingu zařazeném v hrozbě 1 brání vracení falešných hodnot. Tyto hodnoty jsou tvořeny tak, aby co nejvíce připomínaly reálné hodnoty, ale zakryly reálnou identitu a zajistily „zapadnutí do davu“.

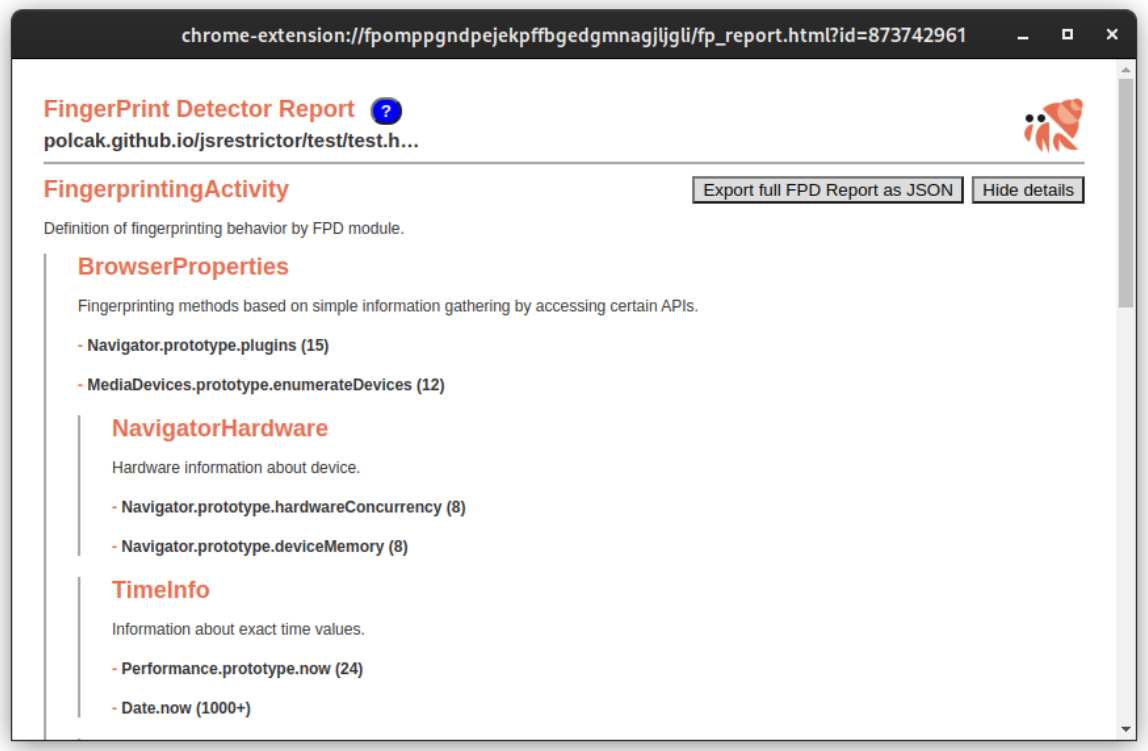
Proti hrozbě 2 JSS implementuje ochranu odstraněním funkcionality některých periferií (např. vrácení úrovně baterie nebo připojené herní zařízení apod.). Pro senzory jako akcelerometr, gyroskop nebo magnetometr vrácené hodnoty emulují stacionární zařízení. Toho je docíleno postupným generováním pseudonáhodných hodnot v čase, podobajících se hodnotám položeného mobilního telefonu.

Dalším typem ochrany je úprava časových značek, zabraňující útokům zařazených do hrozby 4. Toho je docíleno znepřesněním vrácených časových hodnot, buď zaokrouhlením na nastavený počet desetinných míst, nebo vnesením mírné náhodnosti. Přímý přístup k časovým značkám umožňuje velké množství rozhraní, mezi ně patří např. High Resolution Time, Date nebo Performance Timeline, časové značky jsou ale také součástí mnoha objektů vrácených jinými rozhraními. Hrozba 4 je taktéž mitigována pomocí úprav paměťového rozložení polí TypedArray, tato ochrana je však zatím experimentální.

3.2.2 Fingerprint Detector

Fingerprint Detector (FPD) [21] má za úkol analyzovat chování stránky a při zjištění pravděpodobného fingerprintingu informovat uživatele. FPD se skládá ze dvou částí. První je detekce volání funkcí použitelných k tvoření otisků. Ta je přidávána k obalům funkcí tvořených JSS a její princip je předání zprávy o každém volání druhé části FPD. Druhou částí je skript běžící na pozadí, který přijímá zprávy od první části a na základě heuristických postupů počítá pravděpodobnost, že fingerprinting proběhl. V případě, že ano, informuje uživatele notifikací a zobrazí na JShelter ikoně odznak s číslem skupin API, které se o fingerprinting pokusily. Pokud je FPD nastavený jako aktivní, dokáže zablokovat odeslaný požadavek na server s výsledkem fingerprintingu.

FPD také generuje stránku, která uživatele informuje o pokusech tvoření otisku. Tato stránka ukazuje detaily o jednotlivých API, jejich významů pro fingerprinting a počtu jejich volání. Informační stránka je na obrázku 3.1.



Obrázek 3.1: Ukázka výstupu analýzy provedené modulem Fingerprint Detector.

3.2.3 Network Boundary Shield

Modul Network Boundary Shield (NBS) [18] zabraňuje útokům zařazeným v hrozbě 3. Jeho funkce spočívá v blokování požadavků mířených do lokální sítě, čehož dosahuje v různých prohlížečích odlišným způsobem. Prohlížeč Mozilla Firefox podporuje DNS API, NBS je tedy s jeho použitím schopný zjistit zdrojovou a cílovou adresu požadavku ještě před jeho zpracováním. Když zjistí, že se globální adresa pokusila poslat požadavek na lokální adresu, požadavek zablokuje. Prohlížeče založené na jádru Chromium podporu DNS API nemají. V tom případě je pro zjištění síťových adres přijetí a zpracování prvního požadavku nutné. Adresy jsou poté zjištěny z HTTP hlaviček případné odpovědi a na jejich základě jsou blokovány další obdobné požadavky.

3.3 Mechanizmus injekce vlastního kódu

Podmínkou zajištění komplexní ochrany v dynamickém světě internetu je implementace spolehlivého mechanizmu injekce vlastního kódu nutného k provedení potřebných modifikací do prostředí webových stránek. Protože je pro JShelter injekce klíčovou součástí a má významný vliv na výkon načítání stránek, v následující sekci blíže popíši její mechanizmy.

Většinu funkcionality nutnou pro injekci poskytuje knihovna NoScript Commons Library (NSCL)[4], vyvinutá pro účely zjednodušení vývoje bezpečnostních rozšíření a zajištění jejich kompatibility mezi všemi podporovanými prohlížeči. Tato knihovna sice není přímá součást projektu JShelter, ale funkce jí implementované jsou klíčové součásti fungování rozšíření a mají výrazný dopad na jeho výkon, ve zbytku textu tedy původ kódu nebude dál rozlišován, analýza a návrh optimalizací se dotkne i NSCL. Požadavkem pro spolehlivou ochranu uživatele je zajištění obalení objektů a funkcí v kontextu stránky, které musí zaručeně proběhnout před předáním provádění kódu stránce. JShelter z toho důvodu používá dva způsoby injekce.

Prvním z nich je raná injekce (NSCL modul `DocStartInjection.js`), ta spočívá v uložení konfigurace a dynamicky generovaného kódu k injekci v kontextu stránky. Je žádoucí, aby proběhla v procesu načítání stránky co nejdříve, proto je spouštěna (v několika krocích) při různých událostech průběhu navigace a přijímání síťových požadavků. Protože je tvorba konfigurace závislá na výsledcích asynchronních operací a funkce pro vložení a spuštění kódu v kontextu stránky, nutné k provedení injekce, jsou asynchronní taktéž, mohou nastat problémy s její spolehlivostí. Prohlížeče založené na jádru Chromium nepodporují blokující asynchronní zpracování všech potřebných událostí v průběhu příjmu síťového požadavku[5], což znamená, že když při vykonávání takové obslužné funkce interpret narazí na asynchronní volání, je spolu se zbytkem obsluhy zařazeno na konec fronty úkolů k provedení a vykonávání je předáno dalším čekajícím úkolům z fronty. To umožní situaci, kdy je samotné vložení konfigurace a kódu k injekci naplánováno až po provedení ostatních obsahových skriptů rozšíření nebo skriptů stránky, což by způsobilo selhání ochrany. Z toho důvodu raná injekce slouží hlavně jako rychlý, ale nespolehlivý způsob asynchronního načtení konfigurace z pozadí.

Finálním krokem je „ostrá“ injekce, spuštění zavedeného kódu spolu s pomocným kódem pro obalování, jehož součástí jsou například pomocníci pro zakrytí provedených změn a správu pohybu objektů a funkcí na hranici privilegovaného kontextu rozšíření a neprivilegovaného kontextu stránky ve Firefoxu, nebo nástroje pro komunikaci mezi stránkou a rozšířením. Tento krok zahrnuje provedení veškerých potřebných úprav prostředí JavaScriptu a nastává ve skriptu `document_start.js`, zaregistrovaném ke spuštění při načítání každého rámce. Zde se používá standardní injekce obsahových skriptů (angl. *content script injection*) s garantovaným synchronním provedením před kódem stránky či rámce.

Pokud při spuštění ostré injekce není nalezena platná konfigurace v kontextu stránky (raná injekce nestihla proběhnout), musí se vytvořit nyní. Zde opět nastává problém, kdy je nutné získat konfiguraci (tvořenou asynchronně) ze skriptu na pozadí synchronně, jinak hrozí předání kontroly nad vykonáváním stránce. To je řešeno použitím `syncMessage` z knihovny NSCL, což je synchronní obdoba nativních asynchronních zpráv mezi kontexty v prohlížeči. Interně `syncMessage` využívá synchronní požadavek `XMLHttpRequest` na nedosažitelnou adresu. Ten zpracuje obslužná funkce skriptu na pozadí, asynchronně vytvoří konfiguraci ze které sestrojí datové URL a přesměrováním na něj požadavek finalizuje, pozastavený obsahový skript obdržením odpovědi pokračuje. Další komplikaci přináší Google Chrome, který nedokáže obsloužit potřebné události během síťového požadavku na pozadí rozšíření asynchronně a s blokováním. Proto se do dokončení potřebných asynchronních operací na pozadí každé přijetí požadavku zpracuje přesměrováním na stejnou adresu, což způsobí cyklické přijímání stejného požadavku stejnou funkcí pro obsluhu, čímž se efektivně

[4] https://github.com/hackademix/nscl

[5] https://developer.mozilla.org/Add-ons/WebExtensions/Chrome_incompatibilities

zajistí aktivní čekání s předáváním obsluhy asynchronní operaci a pozastavením původního obsahového skriptu.

Součástí pomocníků pro obalování NSCL jsou také kvůli garanci plné ochrany mechanizmy ke sledování DOM objektů stránek, sloužící k zajištění korektního obalení veškerého dynamicky tvořeného obsahu v průběhu celého životního cyklu stránky, včetně plné injekce do všech nezávislých rámců.

Kapitola 4

Analýza výkonu nástroje JShelter

Prvním krokem bylo nainstalování rozšíření JShelter, jeho používání při běžném prohlížení a pozorování jeho vlivu na rychlost načítání stránek a plynulost jejich používání z pohledu běžného uživatele. Nastavení rozšíření bylo ponecháno na výchozích hodnotách – úroveň ochrany JavaScript Shield „Recommended“, zapnutý Network Boundary Shield i Fingerprint Detector.

Již při zběžném pozorování jsem si všiml určitého zpomalení prohlížení. Načtení čistě textové stránky – tedy bez skriptů či rozsáhlých stylů, běžně načtené prakticky bez pozorovatelné prodlevy – nyní mělo pozorovatelnou prodlevu od začátku načítání do použitelnosti stránky. Vliv rozšíření na samotné používání webu ve většině případů nebyl dostatečně velký, aby výrazně ovlivnil běžné prohlížení, ale určité snížení výkonu pozorovatelné bylo. Například weby využívající některou funkcionalitu používanou k otiskování jako HTML Canvas, WebGL nebo manipulaci se zvukem byly výrazně ovlivněny.

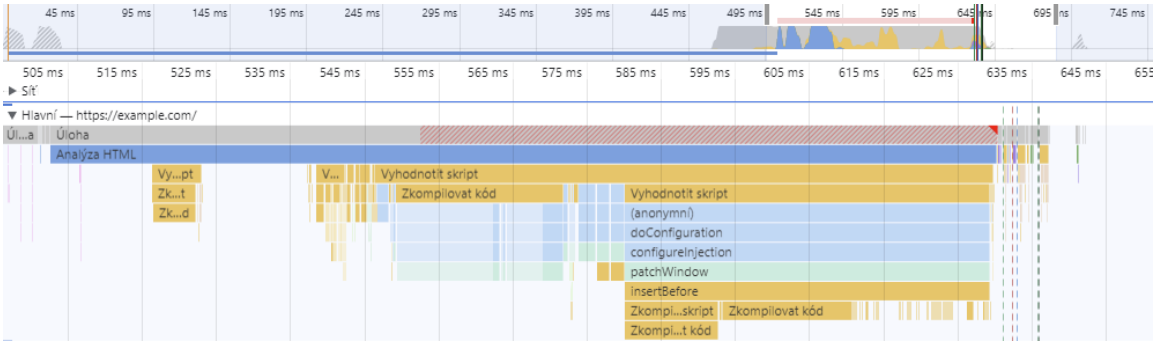
V této kapitole se budu věnovat bližší analýze kritických míst rozšíření JShelter s největším vlivem na výkon při prohlížení využitím nástrojů pro vývojáře v prohlížečích a ruční instrumentací volání. Část 4.1 popisuje chování rozšíření při načítání stránek a vliv mechanizmů injekce na výkon. Část 4.2 se věnuje analýze vlivu ochrany na výkon webových stránek.

4.1 Analýza rychlosti načítání stránek

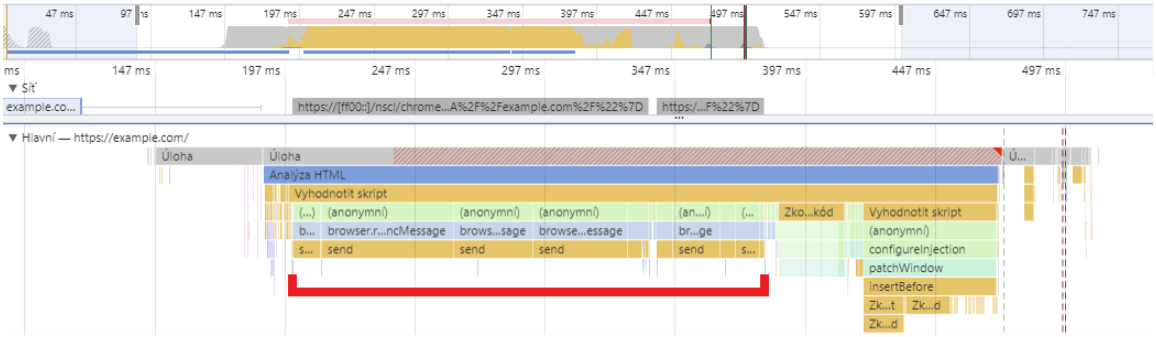
K počáteční analýze vlivu rozšíření na dobu načítání stránek byly zvoleny nástroje pro vývojáře v prohlížeči Google Chrome. Ten byl zvolen, protože jej dle statistik používá majorita uživatelů rozšíření JShelter [20] a dle mých experimentu je použití jeho nástrojů pro profilování a ladění pro mé účely pohodlnější a přesnější. Pomocí nástroje „Výkon“ byl zaznamenán průběh načtení stránky `https://example.com/`. Tato stránka byla vybrána z důvodu její jednoduchosti – měření neovlivňuje načítání dalšího zbytečného obsahu nebo spouštění skriptů, bodem zájmu je čistě chování rozšíření. Nastavení rozšíření bylo opět ponecháno na výchozích hodnotách. Protože Network Boundary Shield nezasahuje do stránek samotných (jeho činností je pouze zpracování síťových požadavků a běží jen na pozadí), lze odvodit, že na rychlost načítání ani prohlížení nebude mít vliv, proto jej nebudu dál při analýze uvažovat.

Dá se očekávat, že výsledky takového profilování nebudou úplně přesné. Nástroje pro vývojáře používají k profilování vzorkování (viz část 2.1.3), tedy sbírají každých 50 μs vzorky obsahující momentálně vykonávané funkce. S narůstající komplexitou programu, použití

asynchronních volání a komunikací mezi skripty zároveň klesá přesnost profilování. Ve výsledcích se například některá volaní ukazují přerušovaně, jako by volání proběhlo vícekrát, to je ale způsobeno nepřesností vzorkování. Taktéž nejsou zaznamenána všechna volání, JShelter při počátečním načtení spouští kód v několika izolovaných kontextech (skripty rozšíření na pozadí, obsahové skripty rozšíření, ale i kód vložený a spouštěný v anonymním kontextu přímo na stránkách) [20], což zabraňuje nástrojům pro profilování mít přehled nad celým vykonávaným kódem.



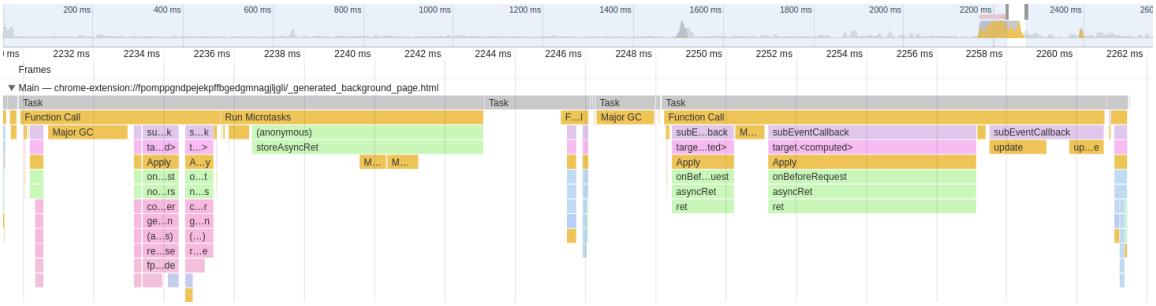
Obrázek 4.1: Výstup analýzy načtení jednoduché stránky, příklad rychlého načtení.



Obrázek 4.2: Výstup analýzy načtení jednoduché stránky, příklad pomalého načtení. Interval, ve kterém probíhá načtení konfigurace, je označen červeně.

Bylo vykonáno několik měření načtením zkoumané stránky se zapnutým nahráváním profilu v nástrojích pro vývojáře, doba načtení byla odečtena ručně z výsledného grafického souhrnu a pro upřesnění byl v některých případech využit výpis přesných časových značek. Výsledky lze klasifikovat do dvou skupin, ve kterých načtení proběhlo vždy téměř ekvivalentně. První skupinou je rychlé načtení, kdy doba běhu skriptu `document_start.js`, vstupního bodu rozšíření a zprostředkovatele injekce obalovacího kódu, byla průměrně 90 ms. Výsledek měření ukazuje obrázek 4.1. Druhou skupinou je pomalé načtení, kdy doba běhu skriptu byla průměrně 250 ms. Výstup je na obrázku 4.2.

Dva velmi odlišné výsledky měření za stejných podmínek jsou způsobeny dvěma odlišnými mechanizmy injekce, blíže popsanými v kapitole 3.3. V případě rychlého načtení raná injekce úspěšně proběhla ještě před předáním obsluhy skriptu `document_start.js` a použije se již dostupný kód vytvořený na základě platné konfigurace, při pomalém načtení je nutné kód a konfiguraci získat synchronním požadavkem ze skriptu na pozadí, který trvá u jednoduchých stránek průměrně 160 ms, což odpovídá času navíc oproti rychlejšímu načtení. Relativní míru frekvence obou případů se z důvodu ovlivnění výsledku nástroji



Obrázek 4.3: Výstup analýzy stránek na pozadí rozšíření při načtení jednoduché stránky, obsluha synchronního požadavku.

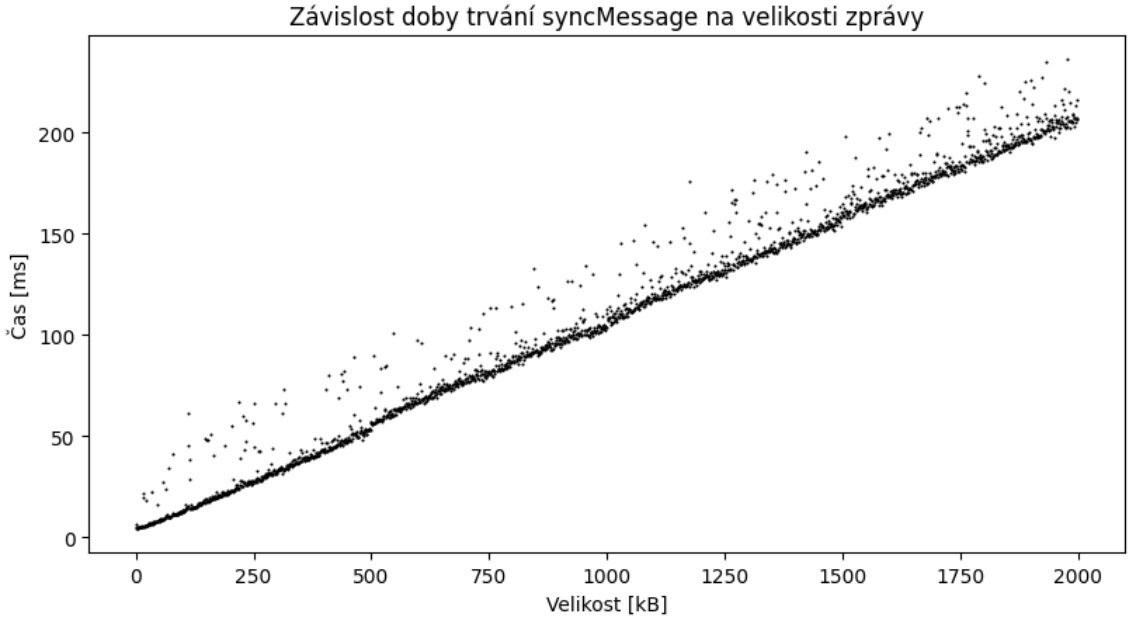
pro vývojáře nepodařilo změřit (při zavedení dalších faktorů je ovlivněno plánování úloh, na kterém je provedení injekce závislé), při experimentech ale na jednoduchých i složitých stránkách téměř vždy proběhla pomalejší injekce.

Důvodem dlouhého trvání `syncMessage` je velká velikost přenášených dat. Kvůli velkému počtu obalů a množství pomocného kódu pro jejich korektní zavedení má kód k injekci ve výchozím nastavení 572 kB. Velikost kódu společně s konfigurací (seznam obalů s aktivními úrovněmi ochrany) zakódovaného jako datové URL, respektive celého požadavku je 706 kB. Profilování stránek na pozadí rozšíření při zpracování požadavku (viz obrázek 4.3) ukázalo, že čas mezi jeho přijetím a odpovědí nepřesahoval 30 ms. Kód k injekci tvořený modulem JSS pro výchozí, neupravené úrovně ochrany je tvořen spolu s inicializací rozšíření, proto je již dostupný. Při použití FPD je nutné kód rozšířit o ještě nepoužité obaly nutné pro fungování modulu, tato operace trvá při obsluze požadavku asi 6 ms. Nejvíce času se při obsluze stráví na serializaci dat k odeslání, operace `JSON.stringify()` a `encodeURIComponent()` zabírají více než polovinu, 15-20 ms, následná deserializace odpovědi v obsahovém skriptu trvá stejnou dobu. Maximální velikost požadavku je omezená na 500 kB, proto se v téměř všech případech použijí požadavky dva a data jsou přenesena postupně, to ale přidává operace pro rozdělení a spojení odpovědi, což způsobí vyšší výpočetní a paměťovou náročnost (která se projeví zvýšením náročnosti garbage collection). Sečtením času stráveného na zpracování požadavku na straně JavaScriptu rozšíření vyjde přibližná hodnota 40 ms, zbylých 120 ms potřebuje prohlížeč.

Pro ověření a posouzení závislosti doby trvání požadavku na velikosti přenesených dat jsem provedl měření, výsledný graf je na obrázku 4.4. Měření probíhalo opakovaným voláním `sendSyncMessage()`, kdy se při každé iteraci uložila doba trvání změřená rozdílem časových značek získaných pomocí `performance.now()`. Každá iterace proběhla jako nová diskrétní úloha, naplánovaná předchozí s využitím `setTimeout` se zpožděním 100 ms. Jako testovací data byl zvolen jednoduchý textový řetězec tvořený opakováním jednoho znaku o velikosti od 1 kB do 2000 kB postupně se zvyšující krokem 1 kB. Z výsledku vyplývá, že rychlost požadavku je lineárně závislá na velikosti přenesených dat. Pro ověření vlivu rozdělení požadavků s velikostí nad 500 kB na rychlost bylo měření zběžně zopakováno bez dělení požadavku a potvrdilo, že dělení na rychlost vliv nemá. Malé „skoky“ v rychlosti však v oblastech kolem míst dělení pozorovatelné byly. Nejrychlejší přenosy při délce zprávy blízké nule trvaly 5 ms a zprávy o velikosti menší než 75 kB byly přeneseny do 10 ms. Ačkoliv naměřené hodnoty neodpovídají hodnotám naměřeným při injekci, jejich kratší trvání přisuzuji zvýšené náročnosti operací při načítání stránky – při měření nebyly prováděny žádné další operace a prohlížeč měl více volných zdrojů na zpracování `syncMessage`. Mě-

ření `syncMessage` bylo zběžně provedeno i v prohlížeči Firefox. Naměřené hodnoty byly velmi podobné (jen s větší variabilitou), proto již nejsou uvedeny.

Při analýze hlavní injekce (zprostředkované funkcí `patchWindow` z knihovny NSCL) bylo nalezeno jedno kritické místo. Tím je volání konstruktoru `new Function` nad celým obalovacím kódem, které vzhledem k jeho velké velikosti trvalo 20 až 30 ms při každém načtení rámce. V prohlížeči Firefox je toto volání nevyhnutelné, kód je dodán jako řetězec a musí být spuštěn v kontextu stránky, v tomto případě přímo v obsahovém skriptu, ale tím je kód již připravený ke spuštění. V prohlížeči Chrome ale obsahové skripty neběží v kontextu stránek, injekce je zajištěna mezikrokem, kdy se kód ke spuštění v kontextu stránky obsahovým skriptem přidá přímo na stránku jako prvek `<script>`. To znamená, že volání `new Function` s převedením zpět na řetězec je z hlediska výkonu zbytečné.



Obrázek 4.4: Výstup měření vlivu velikosti přenesených dat na dobu trvání `syncMessage`.

4.2 Analýza vlivu ochrany na výkon stránek

Dalším krokem bylo ověření vlivu použité ochrany, resp. vlastních obalů, na výkon stránek. Vzhledem k jejich vysokému počtu a výrazným rozdílům v typech a způsobech volání nebylo možné změřit dobu běhu všech obalů a porovnat ji s dobou běhu neobalených volání automatizovaně. Tento problém byl vyřešen ručním procházením všech obalovacích kódů a rozhodováním, jestli obalovací kód přidá výraznou časovou komplexitu vzhledem k typu obaleného objektu a případné odhadované četnosti volání v různých typech aplikací. Seznam rozhraní s výsledky tohoto zkoumání zobrazuje tabulka 4.1.

Většina rozhraní implementuje více než jednu funkci, která představuje riziko z důvodu možného použití při tvoření otisku, stejně tak obal jedné funkce může implementovat více typů ochrany v závislosti na nastavené úrovni. Vysvětlení u každého rozhraní se vztahuje k takové funkci nebo implementaci, která přidá největší časovou komplexnost, tou je myšleno vykonávání dodatečných smyček nad upravovanými daty a složité výpočty.

Rozhraní	Měřit	Důvod
XMLHttpRequest	ne	s přidáním Network Boundary Shield se již nepoužívá
Battery	ne	odstranění funkcionality
Beacon	ne	odstranění funkcionality
Cooperative scheduling	ne	znepřesnění časové hodnoty
Device memory	ne	vrácení falešné hodnoty
DOM	ne	znepřesnění časové hodnoty
TypedArray	**ano**	komplexní znepřesnění velkého množství dat
Date	ne	znepřesnění časové hodnoty
SharedArrayBuffer	ne	zpomalení je součástí ochrany
MediaKeySystem	ne	vrácení falešné hodnoty
Geolocation	ne	asynchronní, vrácení výsledku trvá dlouho i v neobaleném stavu
Gamepad	ne	vrácení falešné hodnoty
Canvas	**ano**	komplexní znepřesnění velkého množství dat
High Resolution Time	ne	znepřesnění časové hodnoty
Worker	ne	zpomalení je součástí ochrany
Window	ne	odstranění funkcionality
HTMLMediaElement	ne	vrácení falešné hodnoty
Idle Detection	ne	vrácení falešné hodnoty
Media Devices	ne	vrácení falešných hodnot
Media Capabilities	ne	vrácení falešných hodnot
NetworkInformation	ne	odstranění funkcionality
NFC	ne	odstranění funkcionality
NavigatorPlugins	ne	znepřesnění vrácených hodnot, malá komplexita
Performance Timeline	ne	znepřesnění časové hodnoty
Sensor	**ano**	komplexní znepřesnění dat
Virtual Reality	ne	odstranění funkcionality
Web Audio	**ano**	komplexní znepřesnění velkého množství dat
WebGL	**ano**	komplexní znepřesnění velkého množství dat
WebXR	ne	odstranění funkcionality

Tabulka 4.1: Seznam rozhraní s kandidáty pro další měření a optimalizaci.

Z prvotního ručního filtrování vyšel seznam rozhraní, na které je nutné se dál zaměřit, protože je pravděpodobné, že zásadně ovlivňují rychlost prohlížení. Tyto rozhraní jsou: TypedArray, Canvas, Sensor, Web Audio a WebGL.

Rozhraní, u kterých bylo rozhodnuto dál nepokračovat s analýzou, dále nebudou uvažovány. Po pečlivém prozkoumání zdrojového kódu jsem usoudil, že časová složitost obalů těchto funkcí již nejde zredukovat z důvodu zachování bezpečnosti a ochrany před otiskováním. Princip ochrany poskytované těmito obaly ve většině případů stojí na odstranění funkcionality nebo navrácení falešné hodnoty. Takové operace jsou samy o sobě velmi rychlé, ale v porovnání s původním nativním voláním (které je součástí prohlížeče, tudíž se vykonává nativní rychlostí a často zahrnuje pouze vrácení reference na existující data) přidávají další vrstvy výrazně pomalejších JavaScriptových volání, kterým se nejde vyhnout.

Jako další krok se nyní nabízí změřit dobu vykonávání funkcí z rozhraní, u kterých je pravděpodobné, že výrazně zpomalují prohlížení webu. Doba vykonávání byla změřena pro

obalené i neobalené, nativní volání. Měření bylo opakováno desetkrát a jako výsledek je uveden průměr z těchto hodnot. JShelter byl ponechán na úrovni ochrany „Recommended“, na této úrovni jsou vykonávány nejnáročnější operace, protože se obaly snaží o úpravu dat bez znehodnocení. Vyšší stupně poskytují ochranu pomocí vracení falešných dat, jejichž vytvoření je méně výpočetně náročné. Měření probíhalo ruční instrumentací volání vybraných funkcí pomocí rozhraní pro vysoce přesný čas.

Jako základ byla použita upravená testovací stránka pro JShelter[1]. Výsledky měření byly ukládány do pole `timing_results`, jehož obsah byl po naplnění odesílán lokálně běžícímu serveru, implementovanému v jazyce Python s využitím knihovny `http.server`[2]. Tento server ukládal průběžně naměřené hodnoty, po desíti nasbíraných hodnotách je zprůměroval a vypsal, výstup ukazuje tabulka 4.2. Funkce pro práci s obrazovými daty byly měřeny na datech o velikosti 960 kB, zvuková stopa měla 176 kB. Měření zkoumaných obalů potvrdilo, že ve většině případů výrazně zpomalí nativní volání, ve zbytku kapitoly se budu věnovat bližší analýze těchto obalů.

Rozhraní	Volání	Délka volání [ms]:	Nativní	Obalené	Zvýšení
Canvas	`isPointInStroke`		0,02	0,42	2000 %
Canvas	`getImageData`		4,73	45,33	858 %
Canvas	`toDataURL`		0,93	49,25	5196 %
WebGL	`readPixels`		3,33	124,17	3629 %
WebGL	`toDataURL`		3	78,42	2514 %
Sensor	`Gyroscope.x`		0	4,33	∞ %
Sensor	`AbsoluteOrientationSensor.quaternion`		0,19	3,21	1589 %
TypedArray	`Float64Array`		1,1	1,1	0 %
TypedArray	`Uint32Array.set`		0,1	0,1	0 %
TypedArray	`DataView.getFloat32`		0,1	0,1	0 %
Web Audio	`AudioBuffer.getChannelData`		0	168,57	∞ %

Tabulka 4.2: Výsledky měření s porovnáním doby obalených a neobalených volání. Naměřené časové hodnoty jsou v milisekundách, zvýšení je vzrůst doby volání v procentech.

4.2.1 Obaly provádějící zpracování dat

Rozhraní Canvas, WebGL a WebAudio poskytují funkce pro vrácení hardwarově generovaných obrazových a zvukových dat zneužitelných při fingerprintingu. Ochrana poskytovaná obaly v tomto případě funguje na stejném principu (*farbling*, blíže popsaný v části 3.2.1), proto je dále budu analyzovat jako celek, jsou to obaly funkcí `getImageData`, `toDataURL`, `readPixels`, `toBlob`, `convertToBlob` a veškeré obaly WebAudio. Obaly funkcí z uvedených rozhraní nevyužívajících farbling (např. `isPointInPath`, získání parametrů hardwarové akcelerace pro WebGL aj.) i přes jejich vliv na výkon z další analýzy vyřazuji, důvodem je jejich konstantní časová složitost, taktéž neobsahují žádné dlouhé operace.

Bylo provedeno profilování použití obalené funkce `Canvas.getImageData` pro získání dat z plátna s 2d kontextem o velikosti 800 na 200 bodů. Samotný farbling probíhá ve funkci `farbleCanvasDataBrave`, výstup jejího profilu je na obrázku 4.5. Ač obal obsahuje i další kód, jeho výkon není oproti výkonu farblingu kritický.

[1] https://polcak.github.io/jsrestrictor/test/test.html
[2] https://docs.python.org/3/library/http.server.html

```
  0.2 ms  let crc = new CRC16();
  0.2 ms  for (row of rowIterator()) {
  0.5 ms      crc.next(row);
          }
          var thiscanvas_prng = alea(domainHash, "CanvasFarbling", crc.crc);
          var data_count = BigInt(BigInt(width) * 4n);

  0.3 ms  for (row of rowIterator()) {
 25.1 ms      for (let i = 0n; i < data_count; i++) {
  7.8 ms          if ((i % 4n) === 3n) {
                      // Do not modify alpha
                      continue;
                  }
  1.5 ms          if (thiscanvas_prng.get_bits(1)) { // Modify data with probability of 0.5
                      // Possible improvements:
                      // Copy a neighbor pixel (possibly with modifications
                      // Make bigger canges than xoring with 1
 90.3 ms              row[i] ^= 1;
                  }
              }
          }
```

Obrázek 4.5: Výstup profilování funkce `farbleCanvasDataBrave` s informací o celkovém čase stráveném na jednotlivých řádcích.

Výstup ukázal, že hlavní důvod poklesu výkonu je iterace nad velkým množstvím dat, v tomto případě je velikost obrazových dat 640 kB (jeden pixel má 4 bajty). Výpočet cyklického redundantního součtu a generace pseudonáhodných čísel zabrala jen malou část volání, nejdéle trvala samotná úprava dat. Překvapivá je doba strávená na vyhodnocení podmínek zanořeného cyklu `for`, které zahrnuje pouze jednoduché operace a běžně probíhá velmi rychle. Vysvětlením je použitý datový typ iterátoru, `BigInt`. Ten umožňuje operace s celými čísly většími než 64 bitů, což je podmíněno jeho nízkým výkonem. Pro ověření byl nahrazen běžným číslem, výsledky opětovného profilování potvrdily nárůst výkonu, čas strávený na vyhodnocení cyklu klesl na 1 ms, na úpravě dat o polovinu.

Protože při použití funkcí pro vrácení obrazových dat lze specifikovat cílový rozsah (v tomto případě souřadnice a velikost vybraného „okna“), farbling musí kvůli zajištění konzistence výsledků proběhnout vždy nad celými daty, nezávisle na výběru. Obaly rozhraní Canvas problém řeší vytvořením kopie původního plátna, na které provedou potřebné modifikace a zavolají původní nativní funkci s originálními argumenty, jejíž výsledek vrátí. Funkce `readPixels` z rozhraní WebGL vrací přečtená data uložením do typovaného pole předaného argumentem, proto stejný přístup (s použitím rychlých nativních funkcí) nelze využít. Řešením je vlastní implementace kopírování upravených dat z pole vytvořeného obalem do pole poskytnutého stránkou funkci dle původního výběru. Měření ověřilo, že oproti samotnému farblingu není výkon této operace kritický, zabírá asi desetinu celkového času výkonu obalu, což je srovnatelné s časem volání nativních funkcí pro stejné účely v obalech rozhraní Canvas (`getImageData`, `putImageData`, případně `toDataURL`).

Analýza volání obalené funkce `readPixels` také odhalila kritický úsek s výrazným vlivem vlivem na výkon nejen tohoto obalu. Volání této funkce na plátně o velikosti 300 na 150 bodů trvalo 83 ms, z toho samotný farbling zabral jen 28 ms. Zbylý čas spotřebovalo odeslání záznamu o volání modulu Fingerprint Detector. To je způsobeno funkcí FPD, který některá volání analyzuje na základě použitých argumentů, tudíž při každém volání je nutné všechny argumenty serializovat a odeslat, což je v případě velkého pole náročná operace. Zpomalení se projeví u jakýchkoliv sledovaných volaní s velkými argumenty.

4.2.2 Obaly rozhraní Sensor

Rozhraní Sensor[3] poskytují funkcionalitu pro čtení dat z fyzických senzorů zařízení. Testování probíhalo na mobilním telefonu OnePlus 8T s použitím prohlížeče Kiwi browser[4]. Tento prohlížeč je založen na technologii Chromium a byl zvolen, protože podporuje načtení rozbalených rozšíření a použití nástrojů pro vývojáře.

V případě rozhraní Sensor jsou obaleny objekty obsahující výsledky měření. Tím je způsobena 0 při měření neobaleného `Gyroscope.x` – `x` je primitivní vlastnost objektu `Gyroscope`, tedy žádné volání neproběhlo. Po aplikaci obalu (princip implementované ochrany popisuje část 3.2.1) se při každém přístupu k jedné z vlastností obaleného objektu nejdříve vypočtou nové hodnoty všech vlastností, což má potenciál ovlivnit výkon při čtení více hodnot výsledku naráz zbytečným přepočítáváním. Při profilování nebyla nalezena žádná konkrétní kritická místa ve výpočtech s výrazným vlivem na výkon, jsou použity matematické operace nad běžnými čísly bez nutnosti iterace.

4.2.3 Obaly rozhraní TypedArray

Rozhraní TypedArray[5] poskytuje typované pole pro práci s pamětí na nízké úrovni. Jejich bezpečnostní hrozba spočívá v mikroarchitekturálních útocích (viz hrozba 4), mitigace je řešena buď použitím pevného posunutí prvků v paměti nebo náhodným přeházením prvků pole v paměti. Při měření nebyl zjištěn rozdíl v době vykonávání neobalených a obalených funkcí pracujících s těmito poli, příčina se vypátrat nepodařila. Vzhledem k principu ochrany bylo očekáváno výrazné zhoršení, přidání dalších volání nutných pro výpočet indexu při každém přístupu do pole má potenciál výrazně negativně ovlivnit výkon. Protože obaly typovaných polí jsou zařazeny do experimentální úrovně ochrany a proveditelnost optimalizace se zachováním veškeré funkčnosti se jeví jako nepravděpodobná, pro další analýzu a optimalizaci nebudou obaly TypedArray uvažovány.

[3] https://developer.mozilla.org/en-US/docs/Web/API/Sensor_APIs
[4] https://kiwibrowser.com/
[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

Kapitola 5

Návrh optimalizací pro JShelter

V této kapitole se budu věnovat návrhu optimalizací pro nástroj JShelter. Optimalizaci injekce, tvorby kódu a načítání stránek popisuje část 5.1, část 5.2 se věnuje optimalizaci obalů provádějících zpracování dat využitím technologie WebAssembly.

5.1 Návrh optimalizace načítání stránek

Měření a analýza načítání stránek ukázaly, že hlavním důvodem nízkého výkonu je rozsah prováděných úprav prostředí stránek. Kvůli zajištění komplexní ochrany je pro každou stránku resp. rámec tvořen rozsáhlý kód k injekci, vložený a provedený v průběhu načítání. Protože tvorba konfigurace (na které je závislá tvorba kódu) je asynchronní proces, připravený kód a konfigurace musí být v průběhu načítání získána velmi pomalým synchronním požadavkem.

Při návrhu byly pečlivě zváženy a prozkoumány možné optimalizace. Zjevnou možností je zmenšení generovaného kódu. V momentálním stavu je pro každý typ obalu generován stejný pomocný kód (pro správné nahrazení vlastností objektů a správu objektů na pomezí privilegovaných kontextu v prohlížeči Firefox), který je ale pro každý obal duplikován. Proto navrhuji tento pomocný kód přesunout do funkcí, které se ve výsledném kódu objeví jen jednou, ale budou moct být volány obaly při jejich zavádění. Jako možnost zmenšení kódu bylo prozkoumáno i zařazení různých metod minifikace (ať už vlastní jednoduché implementace odstranění bílých znaků, komentářů aj. nebo použití opravdového minifikátoru) na konec procesu tvorby kódu, ale bylo posouzeno, že dopad na výkon při použití takových metod bude moc velký.

Kritické místo s největším dopadem na výkon při načítání stránek je `syncMessage`. Protože na výkon zpracování těchto požadavků má většinový vliv samotný prohlížeč, jejich další optimalizace ze strany JavaScriptu se jeví jako nemožné. Pro efektivní optimalizaci musí být buď požadavek při načítání do největší možné míry eliminován, nebo musí být velikost přenesených dat značně snížena. Eliminace `syncMessage` by byla možná zajištěním dokončení rané injekce při každém načtení, to bylo ale zhodnoceno jako neproveditelné kvůli rozdílům implementace zpracování síťových požadavků rozšířeními v prohlížečích, blíže popsaným v kapitole 3.3.

Jako řešení bylo vybráno přesunutí generace kódu ze skriptů na pozadí do obsahových skriptů rozšíření, což eliminuje nutnost převádět pomocí `syncMessage` celý kód, tvořící většinu přenesených dat, čímž se tyto požadavky výrazně zrychlí. Tvorba kódu momentálně probíhá při tvorbě konfigurace, která je závislá na asynchronních operacích pro zápis a čtení

z lokálního úložiště `browser.storage.sync`[1]. Proces tvorby konfigurace tedy nelze kvůli nutnosti synchronního provedení přesunout. Generování kódu však na žádné asynchronní operaci závislé není, potřebuje pouze hotovou konfiguraci.

Přesunutí se nebude týkat pouze funkcí pro generaci kódu, aby mohly fungovat, potřebují znát definice obalů. Zařazení veškerých definic do obsahových skriptů načítaných při načtení každé stránky přinese zhoršení výkonu, kdy musí prohlížeč veškerý kód nejdříve zpracovat. Z měření ale vyplynulo, že zpracování veškerých nově přidaných souborů probíhá rychle. Naměřená doba nepřesáhla 10 ms, což je pro účely optimalizace dostačující a tuto úpravu dělá výhodnou. Samotné vytvoření kódu je (z měření provedeného v kapitole 4.1) také dostatečně rychlé.

Tyto změny umožní výrazně zmenšit velikost dat přenášených `syncMessage` a vkládaných ranou injekcí. To jednak výrazně zrychlí `syncMessage`, také to ale udělá ranou injekci efektivnější a bude schopná častěji úspěšně proběhnout před hlavní injekcí. Rané injekce se týká další návrh zlepšení, a to výměna vlastní implementace algoritmu SHA-256 (použitého k prevenci dvojí injekce stejného kódu vytvořením a kontrolou hashe před provedením) za implementaci nativní, poskytovanou rozhraním `SubtleCrypto`[2]. Z výsledků analýzy v části 4.2.1 vzešlo, že takové zpracování dat je ze strany JavaScriptu pomalé a jeho nahrazení rychlou nativní implementací v prohlížeči poskytne zlepšení výkonu.

5.2 WebAssembly a jeho použití k optimalizaci obalů nástroje JShelter

WebAssembly [16] je relativně nová technologie (oficiálně spuštěná v roce 2017), představující binární formát zdrojového kódu, spustitelného ve všech moderních webových prohlížečích s rychlostí vykonávání blížící se rychlosti vykonávání nativního strojového kódu.

Překlad do formátu WebAssembly je umožněn z velkého množství programovacích jazyků, jejich využitelnost se aktivně vyvíjí a jsou stále přidávány nové[3]. Následující kód je ukázka zkompilovaného strojového kódu WebAssembly, převzatá z oficiálních MDN ukázek[4]. Tento kód poskytuje jedinou funkci `exported_func`, která zavolá libovolnou importovanou JavaScript funkci s jedním parametrem, kam dosadí číslo 42:

```
(module
  (func $i (import "imports" "imported_func") (param i32))
  (func (export "exported_func")
    i32.const 42
    call $i
  )
)
```

Použití WebAssembly obnáší dva kroky. Prvním je načtení zdrojového kódu v binárním formátu (dále vytvořeného z formátu lidsky čitelného kódu z ukázky). Při tomto kroku probíhá validace správného formátu, překlad instrukcí do strojového kódu spustitelného na cílové platformě a načtení spustitelného kódu do paměti. Toto je časově náročný úkon a je nejlepší ho provést jen jednou, při načítání stránky, kde se jeho použití očekává [9]. Druhým

[1] https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage
[2] https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto
[3] https://github.com/appcypher/awesome-wasm-langs
[4] https://github.com/mdn/webassembly-examples/blob/master/js-api-examples/simple.wat

krokem je samotné volání připravených exportovaných funkcí. Toto volání také přidává čas oproti volání regulérních JavaScript funkcí [25], proto se nejvíce vyplatí volat takové WebAssembly funkce, které pracují nad větším množstvím dat, tedy nárůst výkonnosti má dostatek času se projevit.

WebAssembly k chodu využívá lineární blok paměti `WebAssembly.Memory`, který může být modulem jak exportován, tak importován. K jednomu modulu však může být přiřazen maximálně jeden paměťový blok, přiřazení probíhá při jeho inicializaci. K datům v paměti lze přistupovat i z prostředí JavaScriptu, a to vytvořením pohledu na datový blok s využitím typovaných polí.

Pro zrychlení nástroje JShelter navrhuji implementovat optimalizace použitím technologie WebAssembly v obalech rozhraní Canvas, Web Audio a WebGL. Obaly funkcí pro vrácení dat poskytované těmito rozhraními využívají k ochraně farbling, tedy iteraci a zpracování většího množství dat, což je dělá vhodnými kandidáty. Optimalizace bude zahrnovat přepsání kritických funkcí v tabulce 5.1 do jazyka AssemblyScript.

Funkce	Zdrojový soubor
alea	alea.js
crc16	crc16.js
farbleCanvasDataBrave	wrappingL-CANVAS.js
audioFarble	wrappingS-WEBA.js
audioFarbleInt	wrappingS-WEBA.js
farblePixels	wrappingS-WEBGL.js

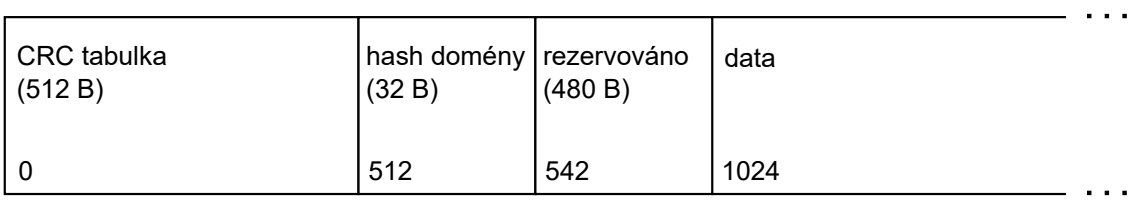
Tabulka 5.1: Seznam funkcí plánovaných k převedení do WebAssembly se zdrojovými soubory, kde se nachází.

AssemblyScript [1] je silně typovaný programovací jazyk v aktivním vývoji se syntaxí převzatou z jazyka TypeScript. Jeho hlavní účel je poskytnutí možnosti kompilovat JavaScript a jemu podobné jazyky do WebAssembly. AssemblyScript implementuje pouze podmnožinu TypeScript funkcionality, což je dané omezeními WebAssembly samotného. Tento jazyk byl zvolen z důvodu podobnosti jeho syntaxe se syntaxí JavaScriptu, to ho dělá žádoucím k účelu přepsání kódu obalů se zachováním stejné funkcionality.

Podmínkou pro fungování optimalizovaného farblingu je sdílení dat mezi WebAssembly modulem a obalem funkce. Protože přenášená data mohou mít velkou velikost, je důležité, aby probíhalo rychle a efektivně. AssemblyScript má podporu komplexních datových typů (v tomto případě by bylo nutné použití typovaných polí) a jeho překladač je schopen vygenerovat kód pro jejich přenos mezi prostředím WebAssembly a JavaScriptem. Použití těchto vlastností ale znamená ztrátu vlastní kontroly nad pamětí, do modulu je zaveden mechanizmus pro správu paměti (*runtime*), který spravuje komplexní objekty v paměti automaticky a zásahy ze strany vývojáře jeho fungování mohou narušit. Taktéž by to znamenalo zhoršení výkonu, přístup k datům přes spravované objekty přidává nutné operace pro indexování v paměti k času vykonávání, přidání runtime zase zvětší velikost WebAssembly modulu, čímž zpomalí jeho inicializaci.

Řešením je vlastní systém pro správu paměti a sdílení dat, což umožní využití rychlých operací pro přímý přístup k paměti a plnou kontrolu nad ní. Návrh schématu paměti je na obrázku 5.1. Pro rychlý výpočet cyklického redundantního součtu a generování pseudonáhodných čísel jsou nutné předpočítané hodnoty (CRC tabulka a hash domény), proto je pro tyto data v paměti rezervován úsek o velikosti 1024 b, který bude naplněný po inicializaci

modulu. Zbytek paměti slouží k uložení dat ke zpracování. Protože je paměť inicializována s předem danou velikostí, před každým voláním modulu musí proběhnout kontrola, zda je kapacita dostačující, a případně se navýší. Kvůli snížení paměťové náročnosti se paměť na začátku bude inicializovat s nejmenší možnou kapacitou, 64 KiB (pevná velikost jedné stránky `WebAssembly.Memory`).



Obrázek 5.1: Návrh rozložení paměti pro WebAssembly modul.

Aby bylo načítání a výkon stránek co nejméně ovlivněno, vyplatilo by se WebAssembly modul inicializovat ve skriptu na pozadí rozšíření. To by ale zahrnovalo nutnost předávat data mezi izolovanými kontexty. Možným řešením je posílání zpráv prohlížeče, protože jsou ale původní obalené funkce synchronní a zprávy prohlížeče fungují pouze asynchronně, tento přístup nelze použít. Nabízí se využití `syncMessage`, ale podle měření její rychlosti v kapitole 4.1 také není efektivním řešením.

Konečný návrh je inicializovat WebAssembly přímo na stránkách, kód pro inicializaci bude součástí pomocníků s obalováním. V prohlížeči Google Chrome injekce vkládá kód přímo na stránku, důsledkem je aplikace jejích pravidel Content Security Policy[5] na vložený kód. Pokud obsahují direktivu `script-src`, inicializace WebAssembly je povolená pouze se zdrojem `'unsafe-eval'` nebo `'wasm-unsafe-eval'`. Proto je nutné přidat blokující obsluhu přijetí hlaviček požadavku, která zajistí přidání zdroje `'wasm-unsafe-eval'` do direktivy `script-src` načítaných stránek, pokud je přítomná. Tento zdroj má vliv pouze na spouštění WebAssembly a protože WebAssembly běží v izolovaném kontextu a potřebuje dodatečný spojovací JavaScript kód k jeho inicializaci a spuštění, takový zásah do CSP neomezí bezpečnost (nebude mít vliv na skripty). Aby byl modul dostupný stránkám, soubor `manifest.json` musí být rozšířen o klíč `"web_accessible_resources"` s cestou k modulu.

Prohlížeč Mozilla Firefox zase přináší odlišnou komplikaci. Kód je při injekci spouštěn v kontextu obsahového skriptu, má plný přístup k objektům stránky a neřídí se CSP stránek, ale je považován za více privilegovaný a kód stránky resp. zavedených obalů s nižšími privilegii nemá přístup k jeho objektům. Rozhraní prohlížeče (rozšířené kódem pomocníků obalování) pro export objektů a funkcí z výše privilegovaných kontextů existuje[6], ale nedokáže exportovat WebAssembly moduly ani paměť (od implementace v prohlížeči neproběhla aktualizace jeho možností, nedokáže si poradit ani s asynchronním programováním na hranici kontextů[7]). Řešením je provádět veškeré operace s pamětí v kontextu obsahového skriptu a výsledná data po farblingu načíst z paměti a exportovat pro užití stránkou.

Kvůli možnému selhání inicializace modulu bude původní JavaScriptová implementace ponechána, inicializace je navíc asynchronní operace, tudíž její dokončení není před spuštěním skriptů stránky garantováno a neexistuje způsob, jak na něj počkat. WebAssembly moduly je sice možné inicializovat i synchronně, ale je to pomalejší a přidalo by to další prodlevu při načítání každé stránky. Aby byla zachována vysoká úroveň ochrany, výsledky

[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
[6] https://developer.mozilla.org/Add-ons/WebExtensions/Sharing_objects_with_page_scripts
[7] https://bugzilla.mozilla.org/show_bug.cgi?id=1757066

obou implementací musí být za každých okolností stejné, jinak by mohl útočník pokus o ochranu detekovat a obejít (bližší popis je v části 3.2.1).

Pro možnost nastavení chování optimalizace uživatelem (při vytváření vlastních úrovní a modifikaci úrovní na specifických doménách) bude přidán nový parametr úrovní ochrany. Kvůli odlišnostem v prohlížečích bude mít tento parametr v obou jiný počet úrovní. V prohlížeči Firefox bude mít dvě úrovně, zapnuto a vypnuto. Pro zajištění inicializace WebAssembly na každé stránce v prohlížeči Chrome je nutné modifikovat přijaté hlavičky CSP, což nemusí být vždy žádoucí (při pokusu o modifikaci stejné hlavičky z více rozšíření se provede jen jedna ze změn, v případě hlavičky CSP se ale změny sloučí a v případě konfliktů se aplikují jen více omezující pravidla[8]). Proto bude mít nastavení tři úrovně, vypnuto, zapnuto bez modifikace CSP (pasivní) a zapnuto s modifikací CSP (aktivní). Kvůli zajištění modifikace hlaviček jen u správných domén bude funkce pro obsluhu požadavků nejdříve vyhledávat v uložených nastaveních pro domény a modifikaci provede pouze, pokud je to pro danou doménu povolené, nebo pokud konfigurace pro doménu neexistuje, ale modifikace jsou povolené v aktivní úrovni JSS. Pasivní optimalizace bude zařazena do výchozí, doporučené úrovně, aktivní optimalizace do experimentální. Ostatní úrovně nepoužívají v rámci ochrany farbling, proto v nich jeho optimalizace povoleny nebudou.

Součástí návrhu je také optimalizace případů, kdy se odesílají modulu Fingerprint Detector zprávy o volání s velkými argumenty (viz část 4.2.1). To bude řešeno úpravou generování kódu, konfigurace obalů se rozšíří o klíč rozlišující typ detekce podle nutnosti rozpoznávání argumentů a kód pro jejich serializaci a odesílání bude generován pouze, když je to nutné. V konfiguraci FPD se nevyskytují žádné sledované volání, u kterých je nutné odesílat argumenty, které mohou být velké. Jako alternativní řešení se nabízí plánování odesílání zpráv pomocí `setTimeout` nebo `queueMicrotask`, což by zajistilo jeho provedení až po dokončení původního volání, ale mohlo by to způsobit prodlevu před obdržením informací modulem FPD a zabránilo vyhodnocení před potenciálním zablokováním odeslání síťového požadavku s vytvořeným otiskem.

[8] https://developer.mozilla.org/Add-ons/WebExtensions/API/webRequest/onHeadersReceived

Kapitola 6

Implementace navržených optimalizací

Sedmá kapitola popisuje vlastní implementaci optimalizací nástroje JShelter navržených v kapitole 5. Změny byly v průběhu implementace publikovány v nové vývojové větvi vytvořené z hlavní větve originálního repozitáře v novém repozitáři na autorově profilu ve službě GitHub[1]. Provedené změny jsou rozděleny do dvou hlavních kategorií. Optimalizace kritických míst při načítání stránek je popsána v části 6.1, zlepšení výkonu obalů – optimalizaci plynulosti stránek – popisuje část 6.2.

Kromě samotných optimalizací byly provedeny některé další úpravy bez vlivu na výkon. Knihovna NSCL byla aktualizována z rok staré na nejnovější verzi, aktualizace přinesla nové funkce a opravy některých chyb, ale použitá funkcionalita a výkon zůstal nezměněn. Také byla opravena chyba v uživatelském rozhraní, kdy v některých případech selhalo načtení konfigurace pro doménu při otevření vyskakovacího okna pro úpravu úrovní na stránce. Zasaženým místem je funkce `connected` nastavená jako obsluha při vytvoření komunikačního kanálu s vyskakovacím oknem v souboru `background.js`. Příčinou chyby bylo špatné volání funkce `tabUpdate`, kde bylo druhým argumentem předáváno URL aktivní záložky, přičemž funkce očekávala objekt předaný při aktualizaci záložky, mající URL jako vlastnost, což způsobilo vrácení nedefinované hodnoty při pokusu o přístup k této vlastnosti. Důsledkem bylo vrácení prázdné úrovně v případě, že URL nebylo nalezeno v mezipaměti ukládající aktivní URL podle identifikátorů záložek `tab_urls`. Řešením bylo načítat konfiguraci pomocí funkce `getCurrentLevelJSON` přímo v obsluze připojení.

6.1 Optimalizace načítání stránek

Tato část se věnuje implementaci navržených optimalizací pro rychlost načítání stránek – zejména zlepšení výkonu generování počáteční konfigurace, obalovacího kódu a jeho injekce. Jako vhodný způsob optimalizace s vysokým potenciálem k výraznému zlepšení výkonu bylo v kapitole 5.1 zvoleno přesunutí generování kódu pro injekci ze skriptů na pozadí do obsahových skriptů, spouštěných v kontextu stránek, což umožní silně zredukovat objem dat posílaný pomalou synchronní zprávou `syncMessage`. V rámci optimalizací načítání byly provedeny i změny ve funkcích pro injekci poskytovaných NSCL a změny formátu přenášené konfigurace.

[1] https://github.com/Martet/JShelter

6.1.1 Knihovna NoScript Commons Library

Změny provedené v NSCL byly v průběhu implementace publikovány stejným způsobem jako změny samotného rozšíření, v nové vývojové větvi na platformě GitHub. NSCL je v repozitáři projektu JShelter zahrnutá jako modul (*git submodule*, odkaz na jiný repozitář v určitém časovém bodě), který byl kvůli inkluzi provedených změn přesměrován z více než rok staré verze původního repozitáře[2] na aktualizovaný a upravený autorův repozitář[3].

Navržené a implementované optimalizace NSCL se týkají pouze prohlížeče Chrome. Pro Firefox se v průběhu analýzy nepodařily najít žádné optimalizace, kritická místa pro výkon jsou funkce provádějící injekci, které jsou v prohlížeči Firefox implementovány „jednodušším“ způsobem a pro Chrome je nutné chybějící funkcionalitu obejít jinak, „složitěji“. Takové řešení ale přináší větší komplexnost a výpočetní náročnost (viz kapitola 3.3 a 4.1).

V souboru `patchWindow.js` bylo upraveno kritické místo, kde se zbytečně převáděl kód pro injekci z řetězce na funkci a vzápětí zpět na řetězec pro vložení na stránku. Samotné odstranění tohoto převodu není dostačující, vložený kód se totiž vzápětí musí zavolat s parametry, a k tomu musí mít správnou signaturu, i když je vložen na stránku jako řetězec. Původní volání:

```
patchingCallback = new Function("unwrappedWindow", "env", patchingCallback)
```

tuto signaturu zajistí. Bylo nahrazeno následovně:

```
patchingCallback = `(unwrappedWindow, env) => {${patchingCallback}}`
```

Tento způsob také garantuje správnou signaturu, ale bez nutnosti zpracování kódu, tudíž je velmi rychlý. Původní konstruktor funkce nutný pro Firefox byl přesunut do pozdějšího úseku funkce `patchWindow`, který se v prohlížeči Chrome nikdy nevykoná.

Ač se tato optimalizace pro účely nástroje JShelter jeví jako vhodná, její použití zanáší do knihovny zranitelnost. Umožní to spuštění kódu bez kontroly syntaxe, což, pokud by útočník získal kontrolu nad kódem předávaným funkci `patchWindow`, může způsobit potenciálně velmi nebezpečný přístup k privilegovanému kontextu rozšíření i veškerému obsahu navštívených stránek. Protože je tento kód vkládán na stránku společně se zbytkem pomocných funkcí pro injekci, je možné sestrojit takový kód, který správně uzavře ohraničující konstrukce funkcí jako jsou závorky, čímž způsobí jeho přístup k okolnímu kódu v souboru `patchWindow.js`. Přítomnost takové zranitelnosti v nástroji JShelter byla vyhodnocena jako bezpečná, nad voláním `patchWindow` nemá útočník jak získat kontrolu (pokud by ji přece jen nějakým způsobem získal, tato zranitelnost by již nejspíš nebyla nutná), přítomnost takové zranitelnosti v knihovně zaměřené na zvýšení bezpečnosti již ale žádoucí není[4].

Druhou implementovanou optimalizací NSCL je nahrazení vlastní implementace algoritmu SHA-256 v rané injekci implementací nativní, poskytovanou prohlížeči. To mírně zvýšilo komplexitu kódu – vstupem i výstupem původní implementace je řetězec, nativní implementace ale pracuje nad binárními daty. Jedná se o funkci `SubtleCrypto.digest`, kterou bylo nutné rozšířit o funkce pro zakódování (zde byl využit nativní `TextEncoder`[5]) a dekódování řetězce (provedeného postupným převedením všech bajtů na znaky). Rozhraní `SubtleCrypto` je dostupné pouze v bezpečných kontextech (při použití HTTPS), tedy na

[2] https://github.com/hackademix/nscl/tree/cead3ec8eabae1638432011335cd914b91124b50
[3] https://github.com/Martet/nscl/tree/25b8650901eb1a51d0cb20e9e9e686cd0fc67432
[4] https://github.com/hackademix/nscl/pull/6
[5] https://developer.mozilla.org/en-US/docs/Web/API/TextEncoder

nezabezpečených doménách nebude fungovat. Protože se ale tato změna dotýká rané injekce, která selhat může, a protože nezabezpečené stránky již nejsou běžné, její ojedinělé selhání je akceptovatelné.

6.1.2 Přesunutí tvorby kódu do obsahových skriptů

Optimalizace s největším pozitivním dopadem na výkon je přesunutí tvorby kódu k injekci ze skriptů na pozadí do obsahových skriptů rozšíření. Veškeré operace nutné k vytvoření kódu jsou synchronně proveditelné v obsahových skriptech, taková změna tedy je možná. Změny v klíčích `content_scripts` a `background.scripts` v souborech `manifest.json` pro oba prohlížeče ukazuje tabulka 6.1.

Nově obsahový	Nově na pozadí

`code_builders.js`
`fp_code_builders.js`
`wrapping*.js`
	`fp_levels.js`

Tabulka 6.1: Seznam zdrojových souborů přesunutých mezi obsahovými skripty a skripty na pozadí. Hvězdička u souboru `wrapping` představuje všechny soubory s definicemi obalů se stejným počátkem jména (celkem 37).

Originální implementace modulu JSS tvoří kód pro vestavěné a uživatelem definované úrovně při načtení rozšíření a uloží ho společně s konfigurací pro každou úroveň v objektu `levels` (tato operace probíhá ve funkci `updateLevels` v souboru `levels.js`), při načtení stránky s takovou úrovní je již připravený a je pouze vrácen. Pokud je pro načítanou stránku uložena úprava úrovně, kód je při načtení znovu tvořen vždy. Kód pro dodatečné obaly FPD se inicializuje stejným způsobem ve funkci `loadFpdConfig` v souboru `fp_code_builders.js` a při načítání jsou do kódu vytvořeného modulem JSS vloženy jen ty obaly, které pro fungování FPD chybí.

Pro přesunutí tvorby kódu bylo nutné veškeré její součásti oddělit od logiky úrovní (která musí zůstat ve skriptu na pozadí). V souboru `levels.js` to zahrnulo odstranění objektu pro uložení kódů `wrapped_codes`, jeho plnění ve funkci `updateLevels` a volání `wrap_code` v případě úprav úrovně s vrácením výsledného kódu ve funkci `getCurrentLevelJSON`, tato funkce nyní vrací jen konfiguraci. Kvůli změně chování `getCurrentLevelJson` byly aktualizovány její jednotkové testy v souboru `tests/unit_tests/tests/levels_tests.js`, žádné další testy upraveny být nemusely, jimi ověřovaná funkcionalita zůstala nezasažena.

Úpravy souboru `fp_code_builders.js` byly rozsáhlejší. Bylo jej nutné rozdělit na dva skripty, `fp_code_builders.js` se změní na obsahový a nově vytvořený `fp_levels.js` zůstane na pozadí. Funkcionalita pro generování kódu z funkce `loadFpdConfig` byla přesunuta do funkce `fp_generate_from_wrappers`, celá funkce `loadFpdConfig` společně s objekty `fp_config_files.js` a `fp_levels` byla přesunuta do `fp_levels.js`.

Objekt `fp_wrapped_codes` byl odstraněn úplně, kód tvořený dynamicky v obsahových skriptech nemá smysl ukládat. Pro zabránění selhání ochrany bezprostředně po načtení rozšíření byl ve funkci `loadFpdConfig` přidán stejný mechanizmus čekání na dokončení inicializace konfigurace jako u úrovní modulu JSS, injekce nyní bezpečně počká na inicializaci obou konfigurací.

V originální implementaci funkce `getContentConfiguration` nacházející se v souboru `level_cache.js` načte konfiguraci a kód, který obohatí obaly modulu FPD. Vše poté buď

pošle synchronním požadavkem skriptu `document_start.js` pro hlavní injekci, nebo předá rané injekci, konfigurace ani kód se již dál nijak nezpracovává, rovnou se použije. Implementované změny toto chování upravují, `getContentConfiguration` vrátí vždy jen konfiguraci, která může být uložena ranou injekci na stránku. Funkce pro tvorbu a obohacení kódu se budou volat při provádění `document_start.js` na základě konfigurace získané buď z kontextu stránky (ranou injekcí), nebo požadavkem `syncMessage`.

6.1.3 Optimalizace velikosti konfigurace a kódu k injekci

Původní implementace přenáší mezi obsahovými skripty a skripty na pozadí objemná data, což se negativně projevuje na výkonu. Odstranění nutnosti přenášet celý kód způsobí největší změnu ve velikosti konfigurace, stále je ale co zlepšovat. Původní formát přenášených dat je následující:

```
{
    currentLevel,   // detaily o použité úrovni
    code,           // kód k injekci
    wrappers,       // seznam použitých obalů JSS
    domainHash,     // hash domény závislý na identifikátoru sezení
}
```

z čehož se použije jen `code` a `domainHash`, ostatní hodnoty se přenášejí zbytečně. Optimalizovaná verze přenáší následující data:

```
{
    currentLevel,   // detaily o použité úrovni a obaly JSS
    fpdWrappers,    // dodatečné obaly pro FPD
    domainHash,     // hash domény závislý na identifikátoru sezení
}
```

kde se všechny hodnoty (kromě některých detailů o úrovni) použijí. Objem přenášené konfigurace tedy vlastně narostl, ale nepřenáší se nic zbytečně, jen narostly požadavky na potřebnou konfiguraci.

Dodatečné definice obalů modulu FPD se v původní verzi ukládají v objektu `fp_levels` pod klíčem `wrappers` jen ve stejném formátu, ve kterém jsou načteny ze souboru `wrappers--lvl_0_1.json`. Pro potřeby tvorby kódu v kontextu stránek ale obsahují zbytečné informace (které jsou využité jen ve skriptu `fp_code_builders.js` na pozadí), proto byl objekt `fp_levels` rozšířen o vlastnost `page_wrappers` obsahující jen konfiguraci potřebnou skriptem `document_start` ve formátu:

```
[
    resource,       // název obaleného objektu
    object_type,    // typ obaleného objektu (0 - funkce, 1 - vlastnost)
    property_type,  // typ vlastností (pole s hodnotami "get" a "set")
    report_args,    // 0 nebo undefined - hlášení argumentů vypnuto
                    // 1 - zahrnout argumenty do zpráv o volání
]
```

Funkce v souboru `fp_code_builders.js` pracující s definicemi obalů byly pro tento formát upraveny.

Kvůli další optimalizaci velikosti výsledného kódu byly v souboru `code_builders.js` provedeny následující změny:

- podmínka pro zmražení obaleného objektu se generuje pouze, pokud je v definici obalu nastavena,
- kód pro vyhledávání v řetězci prototypů původně tvořený pro každý obal zvlášť byl přesunut jako funkce `getDescriptor` do objektu `WrapHelper` a ve finálním kódu se objeví jen jednou,
- kód pro správné nahrazení funkce a správu objektů na pomezí kontextů se generuje pouze v prohlížeči Firefox, jinde je zbytečný.

Pro snadnou identifikaci prohlížeče byla na konec souboru `browser-polyfill.js` přidána proměnná `browser_polyfill_used`, která bude definována pouze, pokud byl tento soubor spuštěn. Protože `browser-polyfill.js` slouží k sjednocení rozhraní prohlížeče Chrome s prohlížečem Firefox, v prohlížeči Firefox není zařazen.

6.2 Implementace optimalizací obalů

Tato část popisuje implementaci optimalizací pro obaly nástroje JShelter poskytující ochranu soukromí uživatele při používání webových stránek. Obaly provádějící zpracování obrazových a zvukových dat (farbling) byly optimalizovány s využitím technologie WebAssembly, její implementace je popsána v části 6.2.1. Část 6.2.2 přibližuje změny, které bylo nutné provést pro integraci nově vytvořeného WebAssembly modulu s rozšířením a jeho obaly. Kvůli možnosti konfigurace této optimalizace uživatelem bylo přidáno nové nastavení a upraveno uživatelské rozhraní, popis těchto změn je v části 6.2.3. Část 6.2.4 osvětluje změny provedené v ohlašování argumentů volání obalených objektů při použití modulu FPD a v části 6.2.5 jsou popsané optimalizace původní, JavaScriptové implementace obalů.

6.2.1 Implementace WebAssembly modulu pro zpracování dat

Z analýzy v kapitole 4.2 vyplynulo, že zpracování obrazových a zvukových dat v obalech rozhraní Canvas, WebGL a WebAudio je velmi pomalé a vyplatí se ho optimalizovat. Jako řešení byla vybrána reimplementace funkcí provádějících farbling do jazyka AssemblyScript, kompilovaném do WebAssembly modulu, který bude přidán jako součást rozšíření a bude zajišťovat optimalizovaný farbling.

V repozitáři byla vytvořena složka `wasm` obsahující zdrojové soubory, konfiguraci a nástroje k překladu. Překladový systém jazyka AssemblyScript je založený na Node.js, proto byl pro WebAssembly modul vytvořen nový Node.js balíček s názvem `wasm_farble`. Pro zajištění automatického překladu a integrace modulu s rozšířením byl aktualizován `Makefile` s příkazy pro překlad (`npm run build`) a zkopírování výsledného WebAssembly modulu do výstupních složek. Modul má jedinou závislost – balíček `assemblyscript`. Samotná implementace se nachází v souboru `farble.ts`, parametry pro překlad nastavuje `asconfig.json`.

Zdrojový soubor obsahuje několik funkcí. Funkce `init` a `next` jsou pro interní použití modulem a slouží k inicializaci, resp. iteraci pseudonáhodného generátoru čísel, dále označovaném jako PRNG (angl. *Pseudorandom Number Generator*). Výstup a stav generátoru je uložen v globálních proměnných. Použití komplexního datového typu (třídy) by pro PRNG z hlediska lepší čitelnosti a udržovatelnosti kódu dávalo větší smysl, třída je ale v jazyce AssemblyScript spravovaný datový typ, tedy bez ztráty vlastní kontroly nad pamětí nelze použít, také by to přineslo zhoršení výkonu při adresaci.

Je důležité, aby optimalizovaná implementace za každých okolností poskytovala stejné výsledky, jako originální implementace v JavaScriptu. Čísla v JavaScriptu jsou reprezentovány dle standardu IEEE 754 pro dvojkovou aritmetiku v pohyblivé řádové čárce s dvojitou přesností (mají velikost 64 bitů, interval bezpečně reprezentovatelných celých čísel je $\langle -2^{53}+1,\ 2^{53}-1\rangle$). Použitý PRNG algoritmus kombinuje standardní aritmetické operace s operacemi pro manipulaci na bitové úrovni. V jazyce JavaScript se takové operace řeší následujícím postupem:

1. desetinná část obou čísel se odstraní,
2. čísla se ořízno na spodních 32 bitů,
3. provede se bitová operace,
4. výsledek se převede zpět a je uložen jako číslo s pohyblivou řádovou čárkou s dvojitou přesností (výsledek ale vždy bude v intervalu $\langle -2^{31}+1,\ 2^{31}-1\rangle \in \mathbb{Z}$).

AssemblyScript používá standardní datové typy s nastavitelnou velikostí se znaménkem i bez, včetně čísel s pohyblivou řádovou čárkou dle standardu IEEE 754, bitové manipulace takových čísel ale standardem definovány nejsou, tedy nejsou podporovány. Proto musel být postup z JavaScriptu implementován i zde. Pro uložení interního stavu PRNG a mezivýsledku při iteraci je použito 64bit číslo bez znaménka a před každou bitovou operací po aritmetické operaci (kde hodnota může opustit interval pro 32bit čísla) se ořízne na spodních 32 bitů. Výsledek generátoru je oříznut a uložen jako globální proměnná s typem 32bit celé číslo bez znaménka.

Implementovaný WebAssembly modul exportuje 5 funkcí pro použití obaly, poskytujících ekvivalentní výsledky s původní implementací, jimiž jsou:

- `crc16(size: usize): u16`

  Vrátí výsledek výpočtu 16bit cyklického redundantního součtu `size` bajtů v paměti na základě předpočítané tabulky v souboru `crc16.js`.

- `crc16Float(size: usize): u16`

  Ekvivalent `crc16` pro `size` 32bit čísel s pohyblivou řádovou čárkou.

- `farbleBytes(size: usize, alea_seed: u32, is_canvas: bool): void`

  Upraví `size` bajtů v paměti na základě počáteční hodnoty `alea_seed`, při předání pravdivé hodnoty parametrem `is_canvas` přeskočí každý 4. bajt (průhlednost v obrazových datech).

- `farbleFloats(size: usize, alea_seed: u32): void`

  Ekvivalent `farbleBytes` pro `size` 32bit čísel s pohyblivou řádovou čárkou.

- `adjustWebGL(x: i32, y: i32, width: i32, height: i32, gl_width: i32, gl_height: i32): void`

  Očekává v paměti již upravená obrazová data (pocházející z plátna s WebGL kontextem) celého původního plátna o velikosti `gl_width` na `gl_height` pixelů. Funkce zkopíruje původní výběr určený souřadnicemi `x`, `y` s velikostí `width` na `height` pixelů na adresu v paměti přímo následující původní upravená data a výběr převrátí po vertikální ose. V případě výběru mimo rozsah originálního plátna jsou chybějící hodnoty

doplněny prázdnými pixely, což je zrychleno uložením hodnot pro prázdný pixel do rezervovaného úseku paměti a jeho opakováním pomocí `memory.repeat` na správných místech výstupního úseku paměti.

6.2.2 Integrace modulu s rozšířením

Pro integraci nově implementovaného WebAssembly modulu s rozšířením JShelter a jeho obaly bylo nutné přidat kód pro jeho inicializaci a zpřístupnění jím exportovaných funkcí do kódu vkládaného na stránky. Toho bylo docíleno vytvořením funkce `insert_wasm_code` v souboru `code_builders.js`, která rozšíří kód generovaný funkcí `generate_code` (kam byla přidána značka cílového místa pro vložení) o požadovaný kód inicializace. Tato funkce je volána v souboru `document_start.js` při tvorbě kódu, pokud je optimalizovaný farbling povolen a nějaká ochrana založená na farblingu je aktivní.

Inicializační kód definuje přednastavené úseky v paměti, vytvoří paměť o nejmenší možné velikosti 64 kB a uloží ji do proměnné `wasm_memory`. Poté zavolá funkci pro načtení, inicializaci a spuštění modulu `WebAssembly.instantiateStreaming`, které předá prázdnou paměť pro propojení s instancí modulu. Funkce je asynchronní, tudíž inicializace neblokuje hlavní vlákno při provádění injekce, což je výhodné pro výkon. Důsledkem ale je, že její konec může nastat až po dokončení injekce a může nastat situace, kdy stránka zavolá obalenou funkci využívající optimalizovaný farbling před dokončením jeho inicializace a musí se použít originální, pomalá implementace. Kvůli stejným výsledkům obou implementací to ale není problém. Binární soubor `farble.wasm` přeložený s konfigurací `release` je velmi malý (má velikost asi 1 kB) a neprovádí po spuštění žádný inicializační kód, tudíž je jeho načtení, přeložení a spuštění rychlé a spotřebuje málo systémových zdrojů. Pro zakrytí webům dostupných souborů používají rozšíření prohlížeče pro takové soubory cestu rozšířenou o náhodně generovaný unikátní identifikátor rozšíření, který lze získat skrze funkci `browser.runtime.getURL` dostupnou pouze kontextům rozšíření. Proto je cesta k souboru získána a do kódu vložena již při jeho tvorbě.

Funkcionalita optimalizovaného farblingu je obalům dostupná skrze zmražený objekt `wasm`. Ten vždy obsahuje vlastnost `ready`, která určuje, jestli byl modul úspěšně aktivován. Počáteční hodnota, nastavená před inicializací, je `false`. Po úspěšném dokončení inicializace je objekt `wasm` rozšířen o funkce exportované modulem, funkce pro kopírování dat a zvětšení paměti a vlastnost `ready` je nastavena na `true`. Po inicializaci se také do rezervovaných úseků paměti zkopírují předpočítaná data – pole `crc16_table` ze souboru `crc16.js` a hash domény převedený z řetězce na 8 32bit čísel bez znaménka.

Pro nastavení dat v paměti slouží `set(data, offset, float)`, kde `data` je objekt typu `UInt8Array` nebo `Float32Array` (což specifikuje logická hodnota parametru `float`) a `offset` je adresa v paměti určující, kam bude celý obsah pole `data` zkopírován.

O vrácení dat z paměti obalům se stará funkce `get(length, offset, float)`, která vytvoří a vrátí objekt `UInt8Array` nebo `Float32Array` (určeno logickou hodnotou `float`) obsahující kopii dat v paměti WebAssembly modulu na adrese `offset` s délkou `length`.

Ověření velikosti a případné zvětšení paměti zajišťuje funkce `grow(needed_bytes)`, kde `needed_bytes` je velikost dat ke zpracovaní v bajtech. Požadovaná velikost se porovná s momentální velikostí a v případě nutnosti se zavolá `wasm_memory.grow` s vypočteným potřebným počtem stránek paměti.

Od samotného WebAssembly modulu a jeho paměti, jejího rozložení a operací s ní jsou obaly odstíněny, veškeré operace provádí skrze zmražené (použitím `Object.freeze` se zakážou veškeré modifikace) rozhraní `wasm`, které je jediným jim dostupným objektem

vytvořeným v rámci inicializace. Hlavním důvodem je bezpečnost – modul i jeho paměť jsou zakryté a není možné provést útok, kdy by se prolomila ochrana prostřednictvím přístupu k modulu ze stránek. Také to zlepší čitelnost a udržovatelnost kódu, funkce v obalech používají jasně definované rozhraní a nemusí řešit adresování v paměti, operace `get` a `set` automaticky přičtou k vyžadované adrese adresu úseku pro uložení dat, čímž znemožní přepsání předpočítaných hodnot v rezervovaných úsecích nutných pro chod modulu obaly.

V prohlížeči Firefox je vložený kód spouštěn jako obsahový skript, tedy modul s připojenou pamětí je inicializován a vlastněn privilegovaným kontextem. To způsobí problémy při převodu dat – obaly jsou exportovány pro použití stránkou a mají nižší privilegia, kvůli čemuž nemají přístup k objektům vytvořeným obsahovým skriptem. Řešením bylo použití funkce `WrapHelper.forPage` na pole vrácené pomocí `wasm.get`. Tato funkce nad daty mimo jiné (řeší i export funkcí, správu asynchronních objektů a ochranu Xray[6]) zavolá funkci `cloneInto`, která z nich vytvoří kopii přístupnou stránce – tedy i obalům – a odkaz na ni vrátí.

Injekce v prohlížeči Google Chrome kód vkládá jako prvek `<script>` přímo na stránky, tedy se inicializace a chod WebAssembly řídí pravidly Content Security Policy stránek, což je popsáno v kapitole 5.2. Proto byla do souboru `background.js` přidána funkce `cspRequestProcessor` pro blokující obsluhu události `onHeadersReceived`. Tato obsluha je nutná jen v prohlížeči Chrome a ve Firefoxu se nepoužije, použitý prohlížeč je zjištěn pomocí vlajky `browser_polyfill_used`. Obsluha při přijetí každého požadavku s obsahem pro hlavní nebo vložený rámec projde všechny hlavičky, když narazí na hlavičku CSP, nahradí řetězec `script-src` za `script-src 'wasm-unsafe-eval'`, čímž umožní spuštění WebAssembly na každé stránce bez změn chování ostatních direktiv nebo zdrojů a pouze v případě, když je to nutné (pokud se `script-src` v hlavičce CSP nenachází, WebAssembly je implicitně povolen a žádné změny nejsou potřeba). Přijatý požadavek je modifikován pouze, pokud byla provedena změna v hlavičce. Taktéž bylo pro Chrome nutné rozšířit jeho `manifest.json` o klíč `"web_accessible_resources"` s cestou k modulu.

Pro integraci se zasaženými obaly byly upraveny následující úseky kódu:

- funkce `farble` v souboru `wrappingS-H-C.js`, přímo zařazená v `helping_code` obalu `getImageData` a nepřímo použitá v obalech `toDataURL`, `toBlob` a `convertToBlob` skrze volání `getImageData`,
- funkce `audioFarble` zařazená v obalech funkcí k načtení zvukových dat s pohyblivou řádovou čárkou a funkce `audioFarbleInt` v obalech k načtení dat v celočíselném formátu v souboru `wrappingS-WEBA.js`,
- funkce `farblePixels` přímo zařazená v obalech `readPixels` (pro obě verze WebGL kontextů) v souboru `wrappingS-WEBGL.js`.

Všechny změny následovaly stejný formát – originální implementace farblingu byla přesunuta do nové funkce uvnitř upravované a nová implementace byla přidána stejným způsobem. Průběh při rozhodování volané implementace je na obrázku 6.1. Obecný průběh při volání funkce s novou implementací farblingu je následující:

1. Voláním `wasm.set` se uloží data ke zpracování do paměti WebAssembly modulu.
2. Zavolá se `wasm.crc16` s velikostí dat, čímž je získán cyklický redundantí součet.

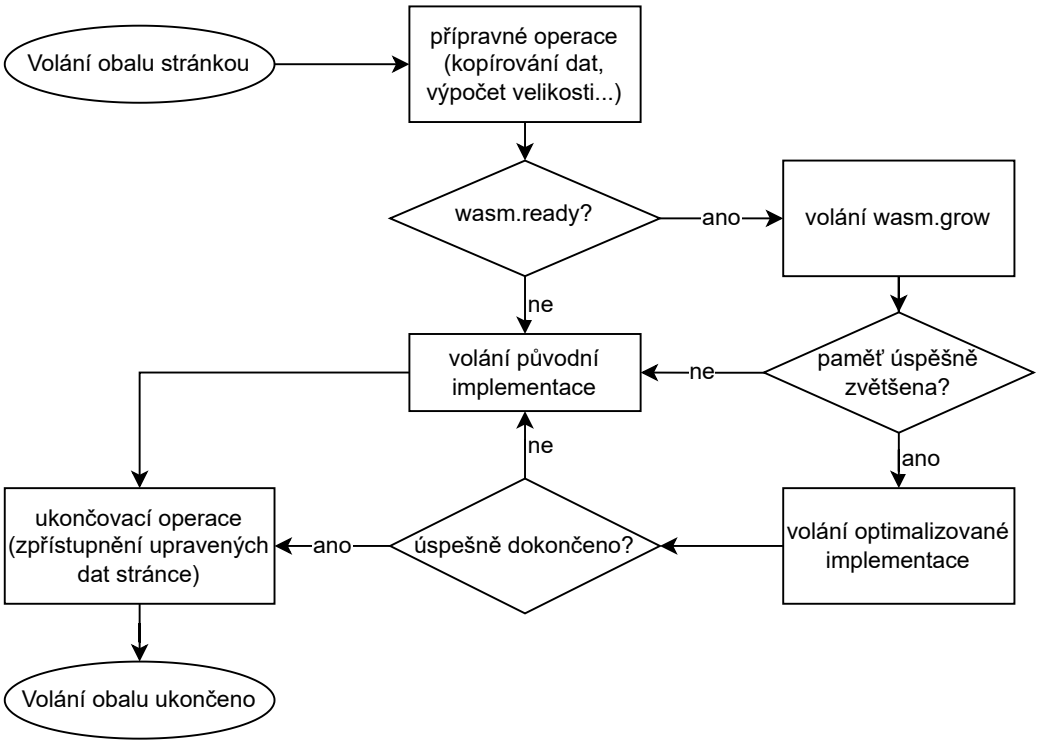
[6] https://firefox-source-docs.mozilla.org/dom/scriptSecurity/xray_vision.html

3. Použije se původní implementace `Mash` pro zkombinování CRC, doménového hashe a názvu momentální operace.

4. Zavolá se funkce pro samotný farbling, která upraví data v paměti.

5. V případě WebGL se zavolá `wasm.adjustWebGL` pro utvoření správného výběru z původních dat.

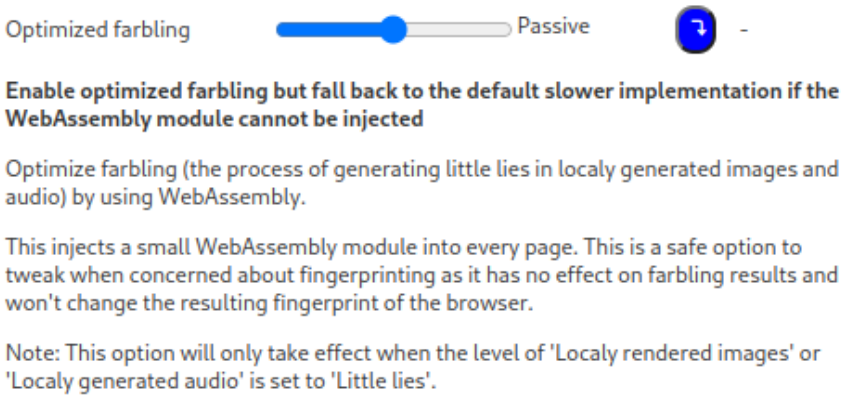
6. Voláním `wasm.get` získá obal kopii upravených dat, kterou správným způsobem vrátí stránce.



Obrázek 6.1: Vývojový diagram popisující postup výběru metody a volání farblingu v obalech poskytujících tuto ochranu.

6.2.3 Implementace uživatelského nastavení pro farbling

Kvůli umožnění nastavení typu použitého farblingu uživatelům byla vytvořena nová položka konfigurace. Definice úrovní `wrapping_groups.groups` v souboru `levels.js` byly rozšířeny o novou skupinu `wasm` s uživatelsky přivětivými popisy a se třemi úrovněmi – vypnuto (0), pasivní (1, zapnuto bez úprav CSP) a aktivní (2, zapnuto s modifikacemi CSP). Protože v prohlížeči Firefox modifikace CSP nutné nejsou, funkce `modify_wrapping_groups` pro úpravu definic úrovní v souboru `levels-browser.js` (načítaném jenom ve Firefoxu) byla rozšířena a při načtení přidané úrovně zredukuje na dvě – vypnuto (0) a zapnuto (1). Toto nastavení bylo přidáno do vestavěné úrovně `Recommended` s hodnotou 1, ale umožní upřesnění konfigurace i pro jednotlivé stránky nebo přidání do uživatelsky definovaných úrovní.



Obrázek 6.2: Snímek obrazovky zobrazující přidané uživatelské nastavení pro optimalizovaný farbling ve vyskakovacím okně pro nastavení stránky.

Pro správné zobrazení tohoto nastavení byly nutné malé změny v kódu pro uživatelské rozhraní. Aby zobrazený název nulté úrovně bylo možné změnit z výchozího „Unprotected“ na něco přesnějšího (toto nastavení úroveň ochrany nemění), ve volání `this.add_tweak_row` ve funkci `tweaks_gui.create_tweaks_html` v souboru `tweaks_gui.js` byl použit předem nevyužitý argument `no_default`, kterému je v případě nastavení WebAssembly předána hodnota `true`. Zobrazení nuly u počtu volání ve vyskakovacím okně pro nastavení na stránkách také není žádoucí (sledování počtu volání optimalizovaného farblingu implementované není), proto byla funkce `popup_tweaks.customize_tweak_row` v souboru `popup.js` rozšířena o podmínku, která v případě tohoto nastavení nahradí číslo za znak „-“. Přidané uživatelské nastavení je na obrázku 6.2.

Nastavení úrovně upravuje chování obsluhy pro modifikaci CSP. Tato obsluha se nachází ve skriptu na pozadí a spouští se nezávisle na tvorbě konfigurace, což znamená, že nemá informaci o nastavení úrovně farblingu pro danou doménu. Proto byla funkce `cspRequestProcessor` rozšířena o kód, který při obdržení každého požadavku ze zdrojové adresy extrahuje subdomény, porovná je s uloženou konfigurací a na základě jí se rozhodne, zda má být hlavička CSP upravena. Pokud není konfigurace pro doménu nalezena, použije se výchozí úroveň.

6.2.4 Zlepšení výkonu při hlášení volání obalených objektů

Při analýze v části 4.2.1 vyšlo najevo, že u volání obalů s velkými argumenty má velký vliv na výkon nutnost je serializovat a odesílat ke zpracování. Bylo zjištěno, že u žádného volání s potenciálně velkými argumenty není nutné jejich sledování, proto byla vybrána optimalizace pomocí označení těch obalů, jichž argumenty je nutné odesílat, a u neoznačených přestat.

Toho bylo docíleno přidáním funkce `fp_append_reporting_to_jss_wrappers` do souboru `fp_code_builders.js`, která se volá v `document_start.js` před vytvořením kódu. Jako argument je předáván seznam obalů FPD s informací, zda je nutné odesílat argumenty. Tato informace se vytvoří při inicializací konfigurace FPD v souboru `fp_levels.js` a je součástí definic obalů FPD pro stránky. Funkce projde všechny definice obalů JSS a u všech, kde je to nutné, přidá klíč `report_args`. Tento klíč je poté využit ve funkci `create_counter_call` v souboru `code_builders.js`, která vygeneruje kód pro serializaci a odeslání argumentů pouze, když je to vyžadováno (hlášení o volání samotném se provede

stále). Pokud modul FPD není zapnutý, seznam jeho obalů je prázdný, tudíž se argumenty nebudou hlásit vůbec.

6.2.5 Optimalizace původní implementace obalů

Měření a analýza odhalily některá další, snadno optimalizovatelná místa s vlivem na výkon obalů se zpracováním dat. V obalech Canvas bylo takové místo nalezeno ve funkci `farbleCanvasDataBrave`, kde se pro indexaci v poli využíval datový typ `BigInt`. Ten umožňuje uložení a výpočty s neomezeně velkými celými čísly, pro zajištění této vlastnosti je však výrazně pomalejší než normální číselný datový typ. V JavaScriptu je největší bezpečně uložitelné celé číslo $2^{53} - 1$, pro uložení plátna s takovou šířkou v bajtech a výškou 1 by bylo potřeba přes 9000 TB paměti. Předpokládám, že tak velké plátno není možné vytvořit, tedy je bezpečné vyměnit `BigInt` za normální číslo bez rizika přetečení indexu.

Druhé nalezené kritické místo se nachází v obalech WebAudio, kde se pro iteraci v poli se zvukovými daty při farblingu používají konstrukce `for (value of array)` a `for (value in array)`. Měřením bylo zjištěno, že taková iterace nad typovanými poli má nízký výkon a spotřebovává hodně systémových zdrojů. Proto byly nahrazeny standardními smyčkami s iterací pomocí číselného indexu v poli, což přineslo výrazné zlepšení výkonu.

Kapitola 7

Zhodnocení provedených změn

Po návrhu a implementaci slibně působících optimalizací je namístě nějak změřit a kvantifikovat vliv těchto změn na výkon vnímaný uživatelem, porovnat naměřené hodnoty s původní implementací a vyhodnotit, jestli se dají provedené optimalizace považovat za úspěšné s přihlédnutím ke všem faktorům jako je například zachování stejné úrovně bezpečnosti nebo uživatelské přívětivosti. Samotná korektní funkčnost všech součástí rozšíření byla v průběhu a na konci implementace optimalizací ověřena testy přiloženými v repozitáři, jimiž jsou testy jednotkové, integrační, systémové a pro FPD. Za účelem ověření nárůstu výkonu byl vytvořen nástroj pro automatické měření uživatelem vnímaného výkonu načítání stránek při použití rozšíření prohlížeče Google Chrome. Tento nástroj ke sbírání dat využívá Google Lighthouse, implementace a analýza naměřených výsledků je popsaná v části 7.1. Vliv optimalizací obalů na výkon stránek byl změřen jako doba trvání těchto volání pomocí ruční instrumentace s využitím rozhraní prohlížeče pro vysoce přesný čas. Postup a výsledky zkoumání jsou v části 7.2

7.1 Analýza načítání stránek s využitím nástroje Google Lighthouse

Pro měření zlepšení výkonu při načítání stránek byl zvolen nástroj Google Lighthouse [5]. Lighthouse je open-source nástroj pro prohlížeč Chrome vyvíjený společností Google a jeho účel je měření výkonu načítání stránek z pohledu uživatele a usnadnění práce vývojářům pomocí sběru dalších užitečných metrik a diagnostik. Lighthouse je integrován v nástrojích pro vývojáře prohlížeče Chrome, je možné ho také spouštět jako nástroj z příkazové řádky nebo ho použít programaticky jako Node.js modul.

Při měření se sbírají následující hlavní metriky:

- První zobrazení obsahu (*First Contentful Paint*, FCP) – čas v ms, kdy se zobrazil první prvek stránky.
- Největší zobrazení obsahu (*Largest Contentful Paint*, LCP) – čas v ms, kdy se zobrazil největší prvek na stránce, aproximuje načtení hlavního obsahu.
- Celkový čas blokování (*Total Blocking Time*, TBT) – celkový čas v ms strávený na úlohách delších než 50 ms (započtená je jen doba nad 50 ms) při načítání stránky.
- Rychlostní index (*Speed Index*, SI) – aproximace doby načítání stránky v ms na základě sledování vizuálních změn.

- Kumulativní změna rozložení (*Cumulative Layout Shift*, CLS) – metrika, která měří intruzivní změny v rozložení stránky při načítání, bez jednotky.

Google Lighthouse následně z těchto metrik počítá váženým průměrem celkové skóre v rozsahu 0–100. Toto skóre má být co nejpřesnější aproximace vnímání rychlosti načítání uživatelem. Váhy jsou nastaveny tak, aby skóre 50 odpovídalo 25. percentilu nejrychlejších stránek a skóre 90 8. percentilu nejrychlejších. Google klasifikuje skóre nad 90 jako výborné, v rozsahu 50–89 jako „potřebuje zlepšení“ a skóre pod 50 jako špatné. Závislost skóre na hodnotách metrik následuje tvar logaritmicko-normální křivky, kde je v rozsahu skóre 50–92 závislost na hodnotách metrik téměř lineární[1]. Toto skóre (dále označované jako výkon) je jediná metrika, kde větší číslo znamená rychlejší, u ostatních hodnot je to naopak.

7.1.1 Implementace automatizovaného nástroje pro měření

Pro pohodlné a automatizované měření byl implementován Node.js balíček využívající nástroj Lighthouse ke sbírání metrik. Pro nástroj byla vytvořena v repozitáři rozšíření JShelter nová složka `tests/performance_tests`. Nástroj tvoří hlavní spustitelný skript `benchmark.js` a konfigurace `config.js`, obsahující výchozí hodnoty pro testování, testované odkazy a konfiguraci nástroje Lighthouse. Nástroj je plně konfigurovatelný parametry příkazové řádky a umožňuje testovat několik rozšíření naráz. Lze nastavit libovolné cesty k rozbaleným sestaveným rozšířením, stejně tak i cesty k souborům `Makefile` a nástroj před testováním rozšíření automaticky sestaví a použije (lze nastavit i implicitní cesta výstupu procesu sestavení). Také je možné rozšíření pojmenovat, jméno se pak objeví místo cesty ve výsledcích, nastavení testovaných URL lze specifikovat souborem `csv` nebo upravením klíče v souboru `config.js`. Pro porovnaní lze měřit i načítání s čistým prohlížečem bez rozšíření.

Výsledek měření je ve formátu JSON, lze ho vypisovat v průběhu testování na standardní výstup, také jde nastavit výstupní soubor, který bude výsledky v průběhu rozšiřován. To umožní znovuobnovení testování při jeho přerušení, ve výchozím nastavení se odkazy již nalezené ve výstupním souboru přeskočí a jsou měřeny pouze ještě nezměřené (parametrem příkazové řádky lze ale toto chování vypnout a soubor vždy přepisovat).

Ke zpracování argumentů nástroj využívá balíček Yargs[2], pro spuštění prohlížeče Google Chrome se správnými parametry se používá balíček Chrome Launcher[3]. Ve výchozím nastavení Chrome Launcher spouští prohlížeč s vypnutou podporou pro rozšíření, proto musel být tento argument z výchozích argumentů odfiltrován. Pro spuštění s rozšířením se využívá argument `-load-extension=<cesta>`.

Nástroj byl koncipován spíše obecněji, proto nefunguje pouze s rozšířením JShelter, ale prakticky s libovolným. Taková implementace byla zvolena hlavně z důvodu, že při průzkumu nebyl nalezený žádný podobný aktuální nástroj. Jediným nalezeným podobným projektem byl Exthouse[4] – funguje na stejném principu, ale již několik let nedostal aktualizaci a již nefunguje s nejnovějším formátem balíčků rozšíření. Proto bylo vyhodnoceno, že bude nejlepší vytvořit vlastní implementaci, oprava několik let starého nástroje by pravděpodobně byla v rychle se vyvíjejícím prostředí webu a webových prohlížečů stejně, ne-li více náročná.

[1] https://developer.chrome.com/docs/lighthouse/performance/performance-scoring/
[2] https://github.com/yargs/yargs
[3] https://github.com/GoogleChrome/chrome-launcher
[4] https://github.com/treosh/exthouse

Hlavní smyčka testování běží ve funkci `runBenchmarks` a silně využívá autorem implementované rozhraní `benchmarkCollector`, jehož funkce slouží k samotnému spuštění měření, uložení a zpracování výsledků. Postup použití nástroje popisuje soubor `README.md`.

7.1.2 Podmínky experimentu

Výkon byl měřen na 50 nejpoužívanějších doménách ze studie Tranco, která agreguje takové seznamy a má za cíl vytvořit seznam co nejpřesnější [11]. Tento seznam agreguje nejpoužívanější domény, ne webové stránky, proto byla ručně odfiltrována asi polovina. Byly odstraněny takové domény, které buď nevedly na žádný web (například servery v sítích pro distribuci obsahu), nebo byly duplikáty (rozdílné domény, které vedly na stejnou stránku, nebo stejné stránky s odlišnou doménou nejvyššího řádu). Všechny domény ve výsledném seznamu se nacházely na nejvyšších 100 příčkách seznamu Tranco. U každé domény byl použit protokol HTTPS, seznam výsledných URL je v příloze B.

Pro každou stránku proběhly 3 měření – čistý prohlížeč bez rozšíření, s původním rozšířením JShelter a s mnou optimalizovanou verzí. Každé měření bylo na každé stránce opakováno 7krát a použil se medián z takto naměřených hodnot. Tato metoda byla vybrána z důvodu zajištění co nejpřesnějších výsledků, měření stránek na internetu v ostrém provozu může být ovlivněno mnoha předem známými i neznámými faktory (dočasné změny v síťové infrastruktuře, změna obsahu stránek, změna v zobrazených reklamách, rozdíly v klientech a mnoho dalších[5]) a naměřené hodnoty se od sebe v některých případech mohou dramaticky lišit. Počítání průměru z naměřených hodnot by mohlo být zásadně ovlivněno v případě, kdy je nějaký z výsledků silně odlišný, ale ostatní jsou si podobné, což použití mediánu řeší [10].

Všechny domény nebyly měřeny v jednom, nepřerušovaném běhu nástroje, běhů s postupným doplňováním výsledků proběhlo několik. Zájmem zkoumání jsou pouze rozdíly v načítání stejné stránky za použití různých rozšíření, měření tedy se zachováním integrity nemuselo proběhnout naráz, všechny měření stejné stránky ale musí proběhnout v co nejkratším časovém intervalu hned za sebou. Konkrétní naměřené hodnoty nejsou kvůli variabilitě výsledků a praktické nemožnosti jejich přesné reprodukce důležité, bodem zájmu je zrychlení, které reprodukovatelné – ač stále s nízkou přesností – je. Každé načtení bylo měřeno jako „studené“, tedy prohlížeč byl mezi každým měřením restartován a byla mu vymazána mezipaměť.

Měření proběhlo na notebooku Lenovo ThinkPad T14s s procesorem Intel i7-10610U a 32 GB paměti RAM se systémem Linux připojeném v nezatížené a spolehlivé domácí síti. Použitá verze Google Lighthouse byla 10.1.1, verze prohlížeče Chrome byla 112.0.5615.165 a původní implementace rozšíření JShelter byla ve verzi 0.12.1, optimalizované rozšíření vycházelo ze stejné verze. Veškeré metody pro simulaci omezení rychlosti sítě a počítače byly vypnuty, předmětem zkoumání je vliv rozšíření na načítání stránek za ideálních podmínek a zanesení dalších ovlivňujících faktorů není žádoucí. Velikost okna prohlížeče při měření byla nastavena na 1920 krát 1080 pixelů v režimu pro počítač. JShelter byl nastavený následovně: JSS byl na úrovni Recommended s upraveným nastavením pro optimalizovaný farbling, které bylo nastavené jako aktivní. FPD byl oproti výchozímu stavu zapnutý, NBS byl zapnutý také, nastavení bylo ponecháno výchozí. Kvůli co nejmenšímu ovlivnění měření ze strany systému byly vypnuty notifikace FPD a NBS.

[5] https://github.com/GoogleChrome/lighthouse/blob/main/docs/variability.md

7.1.3 Výsledky měření

Po provedení samotného měření byly výsledky zpracovány a vyhodnoceny. Naměřené metriky byly pro všechny domény zprůměrovány. Průměr naměřených hodnot pro všechny weby ukazuje tabulka 7.1. Z výsledků vyplývá, že provedené optimalizace vskutku zlepšily výkon načítání stránek, v některých případech signifikantně.

Průměrný výkon stránky s čistým prohlížečem byl 83,56, s původní implementací rozšíření 69,16 a s optimalizovanou verzí to bylo 78,48. To je 13,5% vzrůst průměrného výkonu načítání stránek oproti originální verzi, pro srovnání, vzrůst výkonu při načítání s čistým prohlížečem je 20,82%.

Původní implementace zhoršuje výkon načítání oproti čistému prohlížeči o 17,2 %, zhoršení výkonu s optimalizacemi bylo jen 6,1%, což je zlepšení o 11,1 procentních bodů. Na základě těchto výsledků hodnotím zlepšení průměrného výkonu při načítání jako **velmi dobré**.

	S rozšířením		
Metrika	Původní	Optimalizované	Bez rozšíření
Výkon	69,16	78,48	83,56
První zobrazení obsahu [ms]	1576,17	1348,38	1250,13
Největší zobrazení obsahu [ms]	2056,72	1737,82	1588,28
Celkový čas blokování [ms]	415,47	204,71	114,02
Rychlostní index [ms]	2624,42	2385,31	2264,87
Kumulativní změna rozložení	0,0676	0,0629	0,0706

Tabulka 7.1: Výsledky měření vlivu rozšíření JShelter na výkon načítání stránek posbírané nástrojem Google Lighthouse, průměry z naměřených hodnot pro všechny zkoumané weby.

Nyní by stálo za to blíže prozkoumat dílčí metriky a vyhodnotit dopad provedených optimalizací na ně. Metrika CLS měla zajímavý výsledek – nejhorší hodnota vyšla u prohlížeče bez rozšíření, následovaná výsledkem originální implementace a u optimalizované verze vyšla hodnota nejlepší. Zde by šlo spekulovat, proč tomu tak je. Je možné, že to způsobily změny v době zpracování skriptů na stránkách – při rychlejším zpracování se rychleji vykresloval obsah, tudíž změny v rozložení byly v kratším čase větší. To ale nevysvětluje, proč pomalejší původní implementace dostala horší skóre, než optimalizovaná. Protože jsou ale rozdíly v naměřených hodnotách velmi malé a hodnoty pod 0,1 jsou považovány za dobré[6], tuto metriku nemá smysl dál rozebírat, dalším možným vysvětlením je prostá variablita ve výsledcích.

Metriky FCP, LCP a SI jsou vzájemně propojené – všechny měří čas načtení prvků na stránce. Výsledky vyšly lépe, než bylo původně očekáváno, což je způsobeno načítáním vložených rámců, kde se ve většině případů v originální implementaci také přenášela pomalá zpráva `syncMessage`. Injekce tedy před kompletním načtením vlastně proběhla několikrát a více se projevily provedené optimalizace.

Metrika TBT ukázala největší zlepšení, které také nejvíce ovlivnilo celkové skóre výkonu. Tato metrika má totiž ze všech největší váhu, blokování hlavního vlákna způsobí krátkodobý výpadek responzivity stránky, což uživatel vnímá jako pomalé načítání. Dvojnásobné zlepšení je také způsobeno eliminací velké `syncMessage`, která k této metrice v původní implementaci přidávala signifikantní čas.

[6] https://web.dev/cls/

Celkově by šel vliv implementovaných změn injekce na čas jejího trvání popsat následovně:

- Eliminací velké `syncMessage` bylo ušetřeno 180–200 ms.
- Zařazením definic obalů do obsahových skriptů načítaných při načítání stránek přidalo 10 ms.
- Generování kódu v obsahových skriptech přidalo 15 ms.
- Zrušením zbytečné kompilace kódu při hlavní injekci bylo ušetřeno 25 ms.
- Další menší optimalizace v generovaném kódu ušetřily trochu času a zdrojů počítače.

Tyto hodnoty pochází z měření nástroji pro profilování v průběhu a na konci implementace při načítání jednoduchých stránek a jsou jakýmsi mnou očekávaným zlepšením/zhoršením výkonu na základě pozorování a experimentech při implementaci. Dohromady tyto hodnoty dávají asi 180–200 ms. Takto naměřené hodnoty nemají kvůli možným externím vlivům velkou váhu, jako vysvětlení zlepšení v metrikách ale postačují a odpovídají měření provedenému v této kapitole.

Prohlížeč Mozila Firefox není nástrojem Lighthouse podporován a žádný jiný vhodný ekvivalentní nástroj pro měření výkonu načítání stránek bohužel nebyl nalezen. Možností by byla vlastní implementace takového nástroje, její rozsah ale byl vyhodnocen jako pravděpodobně stejný nebo větší než implementace pro Chrome a musela by výkon hodnotit na základě jiných metrik, sběr zde použitých není podporován. Protože výsledky měření pro oba prohlížeče byly v průběhu analýzy v kapitole 4 ekvivalentní a hlavní provedené optimalizace se mezi prohlížeči neliší, výsledky pro Chrome považuji za platné, ač méně přesné, i pro Firefox.

7.2 Měření výkonu optimalizovaných obalů

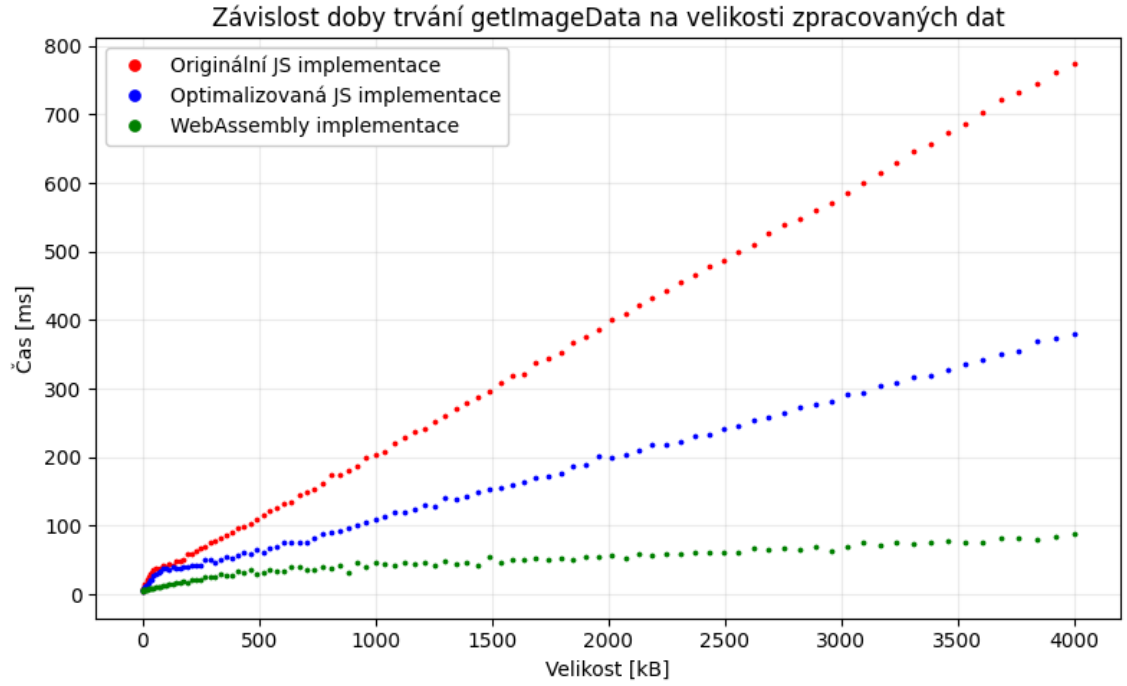
Rozdíly mezi výkonem implementací farblingu, tedy obalů s náročným zpracováním dat, byly zjištěny pomocí měření doby obalených volání s využitím rozhraní prohlížeče pro vysoce přesný čas. K tomu účelu byla vytvořená stránka s potřebnou funkcionalitou, kde byl každý test volán posledním jako diskrétní úloha pomocí `setTimeout` se zpožděním 40 ms a po dokončení běhu se výsledky zprůměrovaly a vypsaly do konzole pro další zpracování. Sledování výkonu v nepřetržité smyčce bez narušení (bez obnovení stránky nebo restartování prohlížeče) bylo zvoleno proto, že optimalizace obalů poskytujících farbling míří hlavně právě na často opakované operace – aby umožnily plynulé použití webových stránek využívající tuto obalenou funkcionalitu za zachování ochrany. Měření proběhlo lokálně na notebooku Lenovo ThinkPad T14s s procesorem Intel i7-10610U a 32 GB paměti RAM se systémem Linux, v prohlížeči Google Chrome 12.0.5615.165. Původní implementace rozšíření JShelter byla ve verzi 0.12.1, optimalizované rozšíření vycházelo ze stejné verze. JShelter měl všechny ochrany, včetně FPD a NBS, kromě farblingu, příp. optimalizovaného farblingu, vypnuté.

7.2.1 Obaly rozhraní Canvas

Prvním z optimalizovaných obalů jsou obaly rozhraní `Canvas`, zasažených funkcí je několik, ale všechny k samotnému farblingu využívají funkci `getImageData`, proto byla měřena právě ta. Počáteční měření proběhlo v prohlížeči Chrome pro čtvercová plátna s kontextem `2d`

o velikosti od 10 krát 10 pixelů do 1000 krát 1000 pixelů s krokem 10, celková velikost zpracovaných dat tedy byla od 0,4 do 4000 kB (jeden pixel má 4 bajty). Takových měření bylo zopakováno 20 a pro každou velikost byl použit průměr ze všech běhů. Výsledný graf pro všechny implementace je na obrázku 7.1.

Graf ukázal téměř lineární závislost doby trvání na velikosti dat, kromě úseku s velmi malými hodnotami. Proto bylo možné se zachováním dostatečné přesnosti spočítat zrychlení mezi implementacemi jako průměrné zvýšení mezi všemi hodnotami dvou lineárních funkcí.

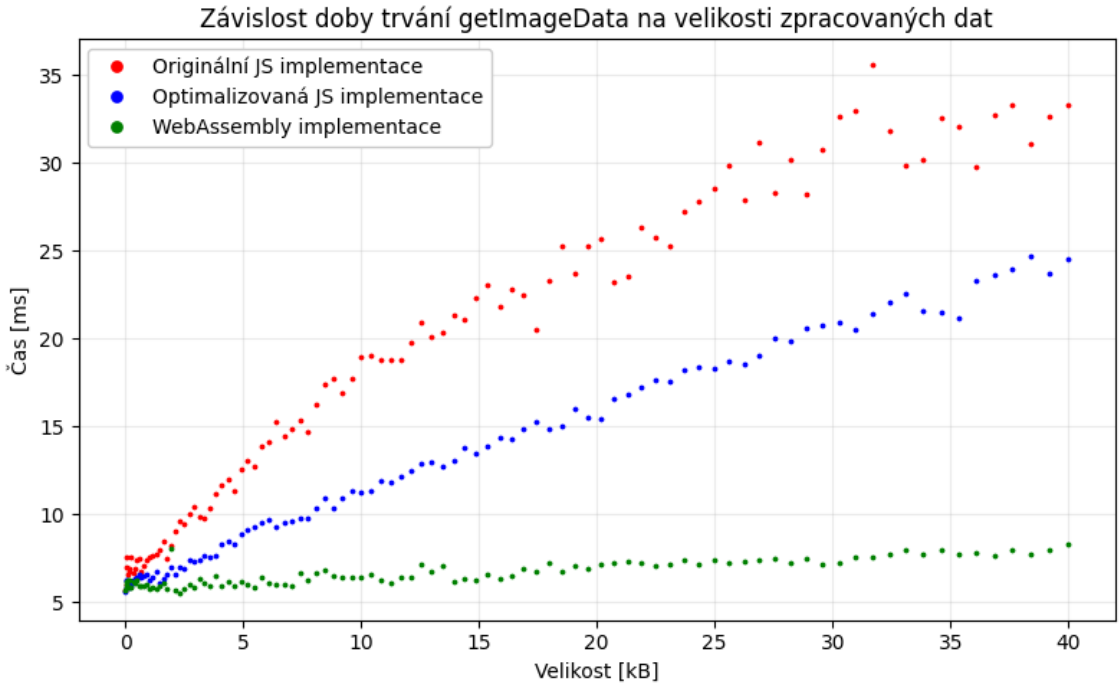


Obrázek 7.1: Výstup měření rychlosti obalené funkce `getImageData` s různými implementacemi pro čtvercové plátno s délkou strany od 10 do 1000 pixelů s krokem 10 (0,4–4000 kB).

Vývoj hodnot při velikostech dat větších než měřených nebyl změřen, použití funkcí na získání dat z tak velkých pláten běžně na webu neočekávám (a původní implementaci trvá zpracování plátna o velikosti 1000x1000 pixelů přes 750 ms, takové volání na stránce má potenciál zpomalit prohlížeč do bodu nepoužitelnosti).

Výsledky jsou následující: optimalizovaná verze obalu `getImageData` v jazyce JavaScript je **1,76krát rychlejší** než původní implementace, využití technologie WebAssembly volání zrychlilo **5,32násobně**.

Pro ověření chování při malých objemech dat bylo měření zopakováno na čtvercovém plátně s délkou strany od 1 do 100 pixelů s krokem 1 (40 B až 40 kB), graf s výstupem je na obrázku 7.2. Toto měření ukázalo, že implementace optimalizovaná technologií WebAssembly je i pro nejmenší možná plátna stejně nebo více výkonná a čas potřebný k volání WebAssembly modulu je dost malý na to, aby se takové volání vyplatilo při každém objemu dat.



Obrázek 7.2: Výstup měření rychlosti obalené funkce `getImageData` s různými implementacemi pro čtvercové plátno s délkou strany od 1 do 100 pixelů s krokem 1 (40 B až 40 kB).

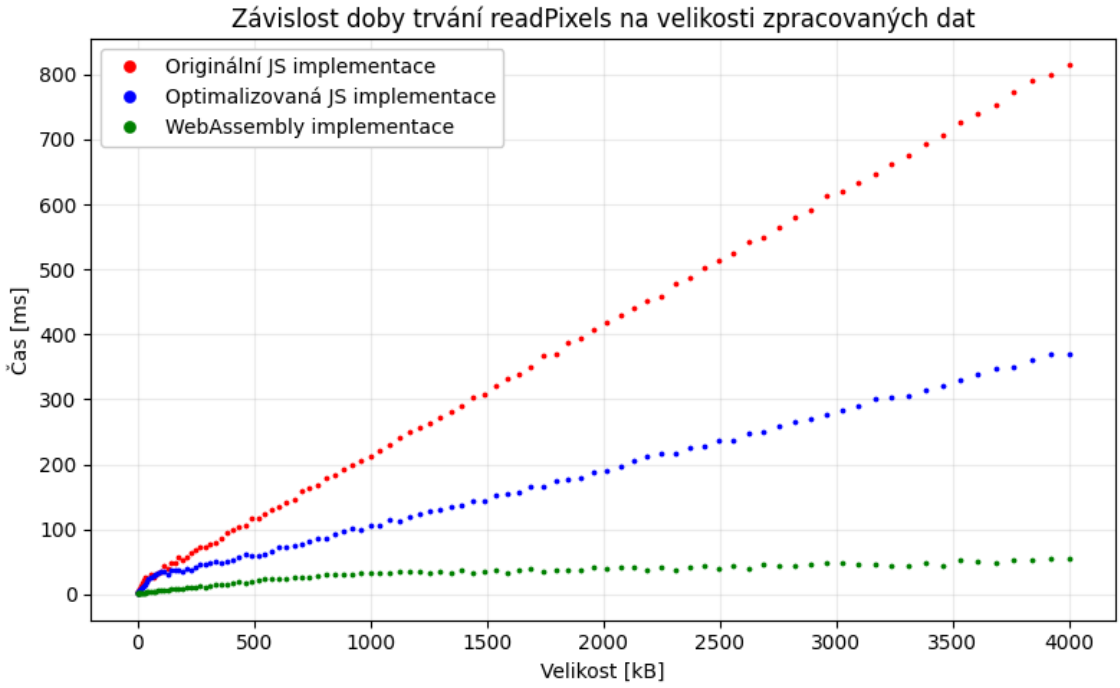
7.2.2 Obaly rozhraní WebGL

Dalším měřeným obalem byl obal funkce `readPixels` objektu `WebGLRenderingContext` z rozhraní `WebGL`. Měření proběhlo za stejných podmínek jako v části 7.2.1. Protože se této funkci předává pole pro uložení výsledných dat argumentem, originální implementaci se nepodařilo stejným způsobem změřit, JShelter začal po několika opakováních kvůli objemu přenášených a ukládaných dat selhávat, vzápětí selhal celý prohlížeč. Proto muselo být z originálního zdrojového kódu pro toto měření odstraněno hlášení volání argumentů. Výstup měření je na obrázku 7.3.

Výsledek je podobný výsledku `getImageData`, průměrné zrychlení pro všechny naměřené hodnoty optimalizované JS verze oproti původní je **1,89násobné**. Optimalizace pomocí WebAssembly přinesla dokonce **8.62násobné zrychlení**, což přisuzuji zrychlení zpětného kopírování původního výběru dat, které je taktéž implementované s využitím WebAssembly (v obalech Canvas se k tomu využívají původní nativní, neobalené funkce).

7.2.3 Obaly rozhraní WebAudio

Z rozhraní `WebAudio` je obaleno 6 funkcí pro získání zvukových dat, všechny obaly fungují na stejném principu, ale řadí se do dvou kategorií – funkce pracující s bajty a funkce pracující s čísly s plovoucí řádovou čárkou. Měřené bylo volání funkce `copyFromChannel` objektu `AudioBuffer` a `getByteFrequencyData` objektu `AnalyserNode`. Výsledky měření `copyFromChannel` jsou na obrázku 7.4, kanál, ze kterého se kopírovalo, byl naplněný náhodnými daty v platném rozsahu $\langle -1, 1 \rangle$.



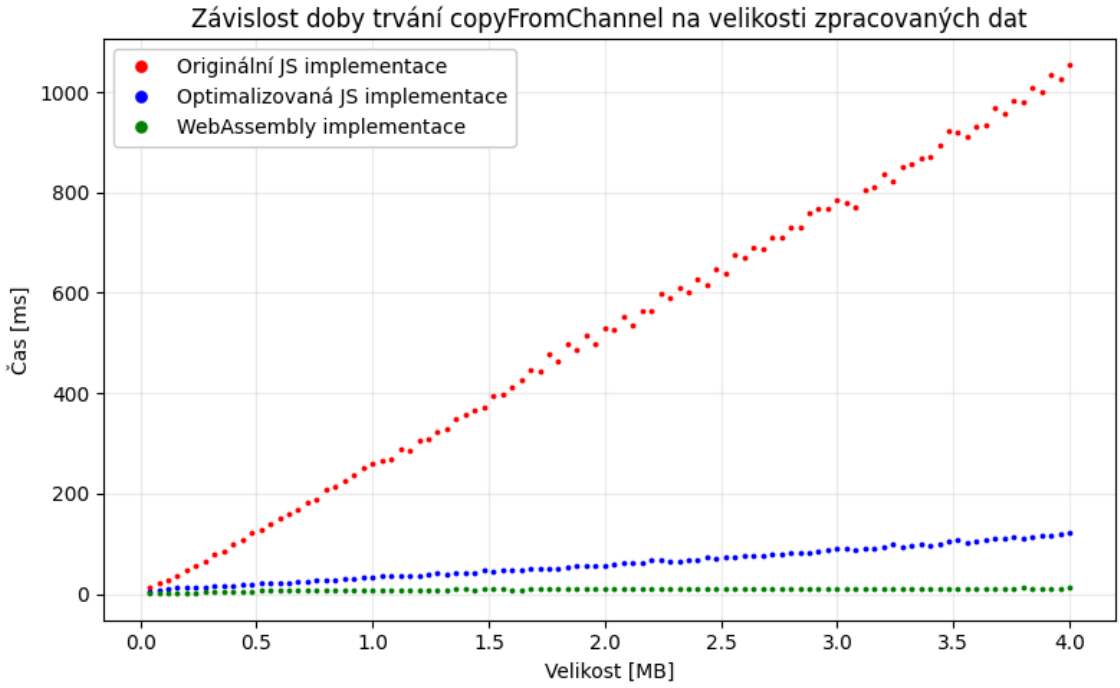
Obrázek 7.3: Výstup měření rychlosti obalené funkce `readPixels` s různými implementacemi pro čtvercové plátno s délkou strany od 10 do 1000 pixelů s krokem 10 (0,4–4000 kB).

Výsledný graf je v tomto případě ze zatím vytvořených „nejhezčí“, je krásně vidět lineární vztah mezi velikostí dat a dobou trvání. Také bylo dosaženo nejlepších výsledků při optimalizaci – optimalizovaná JavaScriptová verze je **8,07krát** rychlejší než originální, WebAssembly tento obal **zrychlil 53.7násobně** a jeho doba volání nepřesáhla v měřeném úseku 12 ms. Takové zrychlení je oproti původní implementaci až neuvěřitelné, vyvozuji z toho, že WebAssembly zvládá výpočty s čísly v plovoucí řádové čárce o několik řádů rychleji než JavaScript.

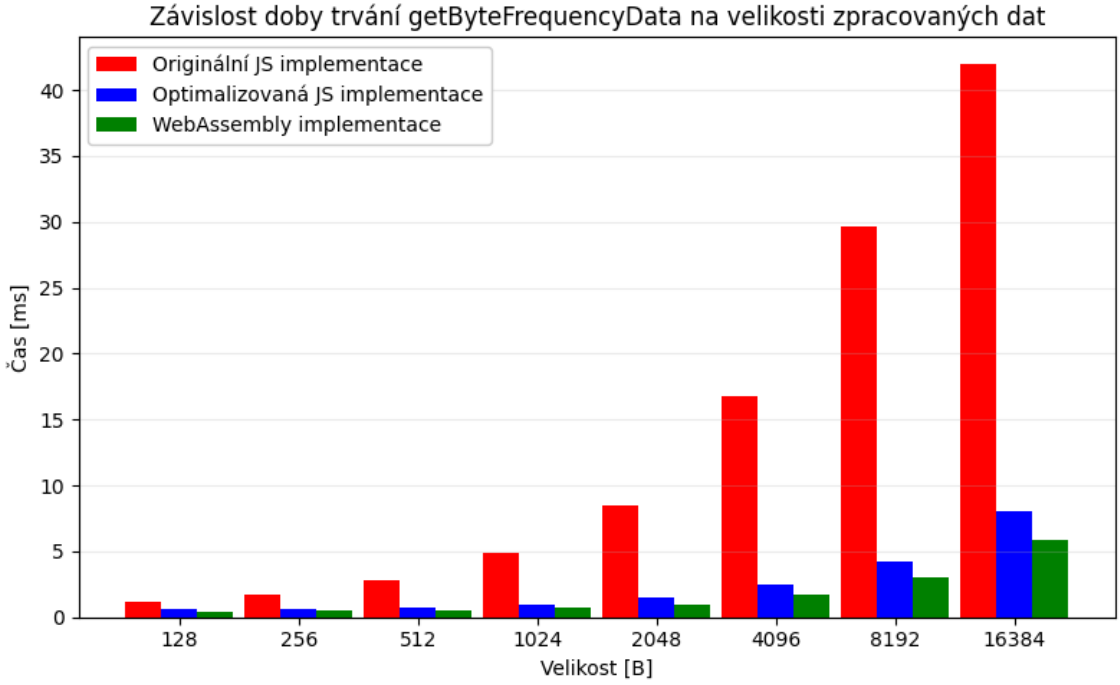
Funkce `getByteFrequencyData` a další obalené, pocházející z objektu `AnalyserNode`, mají odlišnou sémantiku, než doposud měřené obalené funkce. `AnalyserNode`, přeložitelný jako „analyzující uzel“, provádí nad přijatými zvukovými daty rychlou Fourierovu transformaci (FFT) a poskytne informace o frekvenčním složení a časové doméně signálu v reálném čase[7]. Velikost dat zpracovaných obaly se tedy odvíjí od velikosti FFT – pro `getByteFrequencyData` to je poloviční velikost FFT v bajtech, která může být pouze mocnina dvou v rozsahu $\langle 32, 32768 \rangle$, tedy maximální změřitelný rozsah zpracovaných dat je 16–16384 bajtů, 11 možných hodnot. Z toho důvodu bylo opakování měření provedeno 100krát. Výkon obalu této funkce byl měřen bez zavedení vstupních dat – bodem zájmu je chování samotného obalu a poskytnutí dat jeho chování nebo výkon neovlivní, může však ovlivnit výkon samotné funkce, který by pouze vnesl do měření další nejistotu.

Výsledný graf je na obrázku 7.5, první tři velikosti s časem blízkým nule byly pro přehlednost vynechány. V tomto případě – zejména kvůli malé velikosti dat – se optimalizovaná JS implementace a nová WASM implementace lišily ve výkonu jen minimálně, optimalizovaná implementace je však **3,7krát rychlejší** než původní.

[7] https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode



Obrázek 7.4: Výstup měření rychlosti obalené funkce `copyFromChannel` s různými implementacemi pro zvukovou stopu s náhodnými daty o rozsahu 10 kB až 1 MB s krokem 10 kB.



Obrázek 7.5: Výstup měření rychlosti obalené funkce `getByteFrequencyData` s různými implementacemi pro prázdnou zvukovou stopu s velikostmi jako mocniny dvou v rozsahu 128 až 16 384 bajtů.

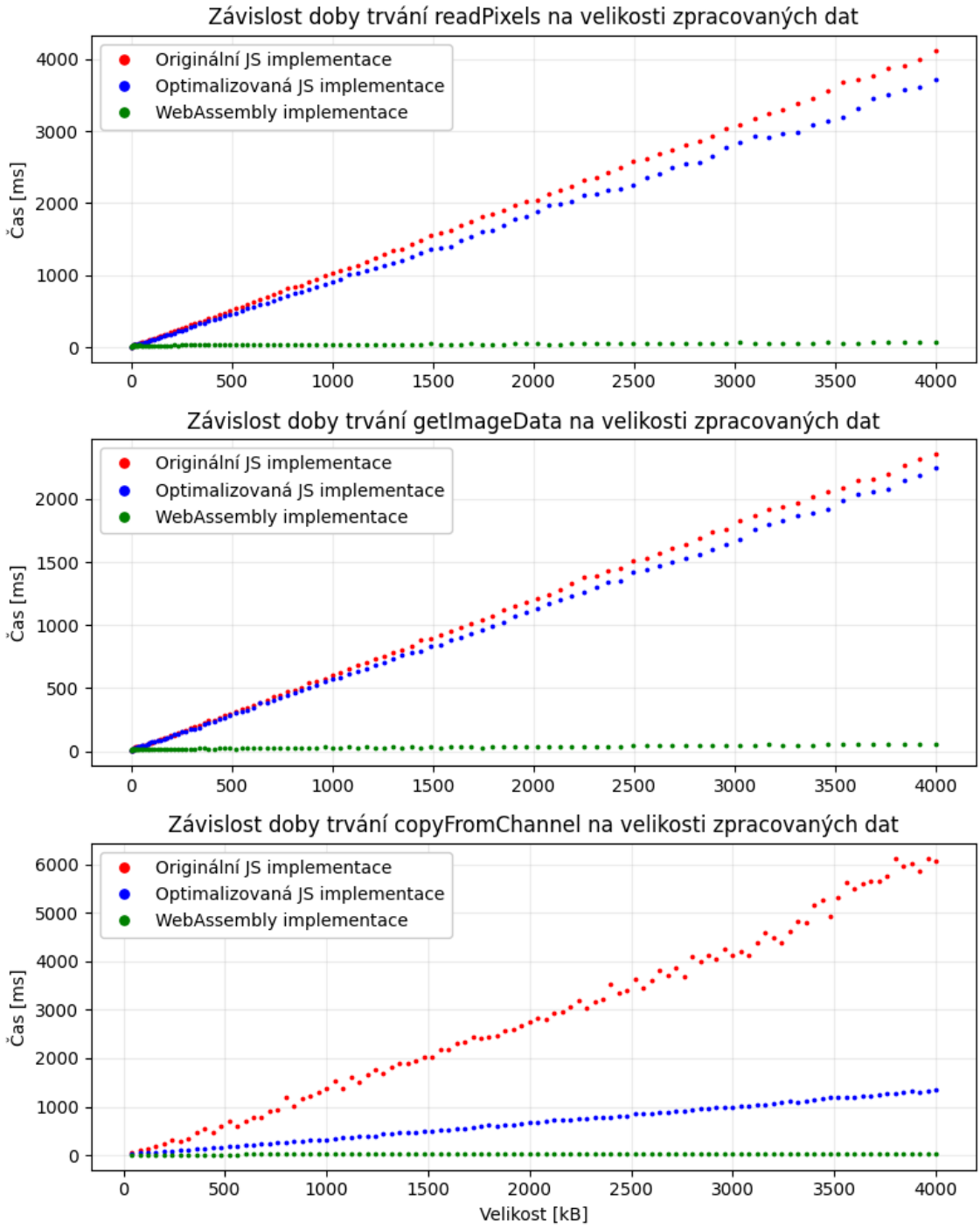
7.2.4 Měření výkonu obalů v prohlížeči Firefox

Pro ověření vlivu jiného JavaScriptového stroje na provedené úpravy bylo stejné měření za stejných podmínek zopakováno i v prohlížeči Mozilla Firefox 102.10.0esr. Tentokrát byly ověřeny obaly tří funkcí, a to `getImageData`, `readPixels` a `copyFromChannel`, tyto obaly byly vyhodnoceny jako nejvíce reprezentativní pro zjištění celkového zlepšení – volají všechny optimalizované funkce využité k farblingu. Výsledek všech měření je na obrázku 7.6.

Oproti prohlížeči Google Chrome se naměřené hodnoty změnily – všechny implementace v JavaScriptu byly výrazně pomalejší, např. volání `copyFromChannel` bylo pomalejší až šestkrát. Funkce `getImageData` a `readPixels` nebyly téměř změnou datového typu iterátoru (`BigInt` na `number`) ovlivněny, jedinou výjimkou byla optimalizovaná verze funkce `copyFromChannel`, která pozorovala změnou typu smyčky výrazné zlepšení, a to asi **čtyřnásobné**.

Takové zpomalení JavaScriptu oproti prohlížeči Chrome je celkem zarážející, podobný výsledek jsem nečekal. Kvůli komplikacím s využíváním jeho nástrojů pro vývojáře a pravděpodobným složitostem v měření a hledání důvodů nebyl tento výsledek dál prozkoumán.

Na druhou stranu, u WebAssembly se projevilo skoro dvojnásobné zrychlení, kdy pozorované hodnoty v měřeném úseku u žádné funkce nepřesáhly 50 ms a jevily se skoro jako konstantní. WebAssembly přineslo ještě výraznější zrychlení, což v kombinaci s možností jeho volné inicializace na všech stránkách bez zásahů do hlaviček hodnotím jako velmi úspěšnou optimalizaci s pozitivním vlivem na výkon se zachováním stejné úrovně ochrany.



Obrázek 7.6: Grafy zachycující výsledky měření rychlosti obalených funkcí `getImageData`, `readPixels` a `copyFromChannel` s různými implementacemi v prohlížeči Firefox pro rozsah dat 10 kB až 4000 kB.

Kapitola 8

Závěr

Cílem této práce bylo zmírnit dopad nástroje JShelter na výkon při prohlížení webu, čehož bylo docíleno návrhem a implementací vlastních úprav, které přinesly pozorovatelné zlepšení. Při řešení jsem se seznámil se způsoby měření výkonosti programů, s rozšířením JShelter, principem jeho fungování a s hrozbami, proti kterým chrání. JShelter jsem poté profiloval přesným časováním důležitých volání a použitím nástrojů pro vývojáře v prohlížečích. Na základě výsledků analýzy jsem navrhl a implementoval optimalizace nalezených kritických míst, jejichž efektivitu jsem ověřil časováním zasažených volání a pomocí metrik poskytnutých nástrojem Google Lighthouse.

Z analýzy vyplynulo, že rozšíření JShelter zpomaluje vykonávání všech volání, která obaluje, některá až o tisíce procent. Například volání pro získání datového odkazu na obsah plátna, běžně provedené v rámci jednotek milisekund, se zpomalilo 500krát. Taktéž jsem zjistil, že JShelter ovlivňuje načítání stránek, kam přidá prodlevu o trvání 200–250 ms a průměrný výkon při načítání v prohlížeči Google Chrome zhorší o 17,2 %. Proto jsem implementoval optimalizace s cílem tato místa v největší možné míře se zachováním stávající vysoké úrovně ochrany eliminovat. Funkce vnášející nepřesnosti do obrazových a zvukových dat poskytovaných zranitelnými rozhraními byly optimalizovány s využitím technologie WebAssembly, která umožňuje takové zpracování dat provádět velmi rychle, např. při práci se zvukovými daty to přineslo více než padesátinásobné zrychlení. Původní implementace se podařila zrychlit také, a to použitím výkonnějších datových typů a efektivnějšími způsoby iterace, zde bylo zrychlení až osminásobné.

Načítání stránek bylo zrychleno pomocí úprav mechanizmů pro získání konfigurace pro stránku, vkládání vlastního kódu a zavedení ochrany přesunutím tvorby kódu do obsahových skriptů. Tyto optimalizace byly efektivní, průměrný výkon načítání vybraných měřených stránek vzrostl o 13,5 % a zhoršení oproti čistému prohlížeči bylo jen 6,1%.

Další práce bude záviset na nejisté budoucnosti specifikace Manifest V3 v prohlížeči Chrome pro rozšíření. Google v budoucnu plánuje zrušit podporu rozšíření využívajících starou verzi, datum přechodu bylo ale několikrát posunuto a již není ani uvedeno. JShelter je momentálně závislý na některých funkcích specifikace Manifest V2, které budou ve V3 odstraněny. Řadí se mezi ně např. funkce pro vykonání kódu uloženého jako řetězec nebo synchronní síťový požadavek. Pro zajištění funkčnosti rozšíření v budoucnosti tedy budou pravděpodobně nutné signifikantní změny v architektuře. Pokud budou tyto úpravy navrženy s přihlédnutím k výkonu, mohlo by to přinést výrazné zlepšení. Nabízí se např. přesunutí funkcí pro zavedení ochrany do obsahových skriptů, pro které by mohl být vytvořen systém dynamické injekce při načítání, pokud by taková implementace se zachováním garance injekce byla možná.

Literatura

[1] AssemblyScript authors. *The AssemblyScript book* [online]. 2020 [cit. 2023-04-25]. Dostupné z: https://www.assemblyscript.org/.

[2] Bergbom, J. *Attacking the internal network from the public Internet using a browser as a proxy* [online]. Research report. Forcepoint, březen 2019 [cit. 2023-05-05]. Dostupné z: https://www.forcepoint.com/sites/default/files/resources/files/report-attacking-internal-network-en_0.pdf.

[3] Bortz, A. a Boneh, D. Exposing private information by timing web applications. In: Association for Computing Machinery. *WWW '07: Proceedings of the 16th international conference on World Wide Web*. New York, NY, USA: [b.n.], Květen 2007, sv. 16, s. 621–628. ISBN 978-1-59593-654-7.

[4] Flanagan, D. *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language*. 7. vyd. Sebastopol, CA, USA: O'Reilly Media, květen 2022. ISBN 978-1491952023.

[5] Google developers. *Lighthouse documentation* [online]. 2016 [cit. 2023-05-06]. Dostupné z: https://developer.chrome.com/docs/lighthouse/.

[6] Graham, S. L., Kessler, P. B. a McKusiclc, M. K. Gprof: A call graph execution profiler. In: Association for Computing Machinery. *ACM SIGPLAN Notices*. New York, NY, USA: [b.n.], červen 1982, sv. 17, s. 120–126. DOI: 10.1145/800230.806987. ISSN 0362-1340.

[7] Grigorik, I. *High Resolution Time Level 2* [online]. W3C Recommendation. World Wide Web Consortium, listopad 2019 [cit. 2023-05-05]. Dostupné z: https://www.w3.org/TR/2019/REC-hr-time-2-20191121/.

[8] Gruss, D., Maurice, C. a Mangard, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: Association for Computing Machinery. *International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. červenec 2016, sv. 13, s. 300–321. ISBN 978-3-319-40666-4.

[9] Haas, A., Rossberg, A., Schuff, D., Titzer, B., Holman, M. et al. Bringing the web up to speed with WebAssembly. In: Association for Computing Machinery. *The 38th ACM SIGPLAN Conference*. New York, NY, USA: [b.n.], červen 2017, s. 185–200. DOI: 10.1145/3062341.3062363. ISBN 978-1-4503-4988-8.

[10] Heričko, T., Šumak, B. a Brdnik, S. Towards Representative Web Performance Measurements with Google Lighthouse. In: University of Maribor. *7th Student*

Computer Science Research Conference (StuCoSReC). Maribor, Slovenia: [b.n.], Září 2021, s. 39–42. DOI: 10.18690/978-961-286-516-0.9. ISBN 978-961-286-516-0.

[11] Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M. a Joosen, W. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In: The Internet Society. *Proceedings of the 26th Annual Network and Distributed System Security Symposium*. San Diego, CA, USA: [b.n.], únor 2019. NDSS 2019. DOI: 10.14722/ndss.2019.23386. ISBN 1-891562-55-X.

[12] Lerner, A., Simpson, A. K., Kohno, T. a Roesner, F. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In: USENIX Association. *Proceedings of the 25th USENIX Security Symposium*. Berkeley, CA, USA: [b.n.], Srpen 2016, sv. 25, s. 997–1013. ISBN 978-1-931971-32-4.

[13] Liang, S. a Viswanathan, D. Comprehensive Profiling Support in the Java Virtual Machine. In: Advanced Computing Systems Association. *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems*. San Diego, CA, USA: [b.n.], Květen 1999, sv. 5, s. 229–242. ISBN 1-880446-35-9.

[14] Malony, A. D. a Huck, K. A. General Hybrid Parallel Profiling. In: IEEE Computer Society. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Washington, DC, USA: [b.n.], únor 2014, sv. 22, s. 204–212. DOI: 10.1109/PDP.2014.38. ISBN 978-1-4799-2729-6.

[15] Mayer, J. R. a Mitchell, J. C. Third-Party Web Tracking: Policy and Technology. In: IEEE. *2012 IEEE Symposium on Security and Privacy*. Květen 2012, sv. 33, s. 413–427. DOI: 10.1109/SP.2012.47. ISBN 978-1-4673-1244-8.

[16] MDN contributors. *WebAssembly* [online]. Mozilla Corporation, 2022 [cit. 2023-05-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly.

[17] Nägele, T. *Client-side performance profiling of JavaScript for web applications*. Nijmegen, NL, 2015. Diplomová práce. Universidad de Radbound.

[18] Pohner, P. *Detekce podezřelých síťových požadavků webových stránek*. Brno, CZ, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22377/.

[19] Polčák, L., Maone, G., Saloň, M., Hranický, R. et al. *JShelter Website* [online]. 2022 [cit. 2023-05-05]. Dostupné z: https://jshelter.org/.

[20] Polčák, L., Saloň, M., Maone, G., Hranický, R. a McMahon, M. *JShelter: Give Me My Browser Back* [online]. arXiv, duben 2022. revidováno 1. června 2022 [cit. 2023-05-05]. DOI: 10.48550/ARXIV.2204.01392. Dostupné z: https://arxiv.org/abs/2204.01392.

[21] Saloň, M. *Detekce metod zjišťujících otisk prohlížeče*. Brno, CZ, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/23645/.

[22] Smith, M., Disselkoen, C., Narayan, S., Brown, F. a Stefan, D. Browser history re:visited. In: USENIX Association. *Proceedings of the 12th USENIX*

Conference on Offensive Technologies. Berkeley, CA, USA: [b.n.], Srpen 2018, sv. 12, s. 16. ISBN 9781713804512.

[23] Thiel, J. *An overview of software performance analysis tools and techniques: From gprof to dtrace*. Washington University in St. Louis, Tech. Rep, 2006.

[24] Vysocký, O., Říha, L. a Bartolini, A. Application instrumentation for performance analysis and tuning with focus on energy efficiency. In: John Wiley & Sons Ltd. *Concurrency and Computation: Practice and Experience: Special Issue to 13th International Conference on Parallel Processing and Applied Mathematics (PPAM2019)*. Květen 2021, sv. 33, č. 11. ISSN 1532-0626. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5966.

[25] Yan, Y., Tu, T., Zhao, L., Zhou, Y. a Wang, W. Understanding the Performance of Webassembly Applications. In: Association for Computing Machinery. *Proceedings of the 21st ACM Internet Measurement Conference*. New York, NY, USA: [b.n.], 2021, s. 533–549. DOI: 10.1145/3487552.3487827. ISBN 9781450391290. Dostupné z: https://doi.org/10.1145/3487552.3487827.

Příloha A

Obsah datového nosiče

Přiložené datové CD obsahuje následující složky a soubory:

- `JShelter` — repozitář se zdrojovým kódem nástroje JShelter s provedenými optimalizacemi, nástrojem pro měření výkonu pomocí Google Lighthouse a výsledky tohoto měření,
- `NSCL` — repozitář knihovny NSCL s provedenými optimalizacemi,
- `Sestavené verze` — sestavené verze nástroje JShelter s provedenými optimalizacemi pro oba prohlížeče v rozbalené i zabalené podobě,
- `Stránky pro měření a testování` — upravené stránky projektu JShelter pro měření a testování,
- `Technická zpráva` — technická zpráva a její zdrojové soubory,
- `README.md` — soubor s podrobných popisem obsahu datového nosiče.

Zdrojový kód je také publikován na autorově GitHub profilu:
https://github.com/Martet/JShelter
https://github.com/Martet/nscl

Příloha B

Seznam testovaných URL

Pozice	URL	Pozice	URL
1	https://google.com	40	https://mail.ru
2	https://facebook.com	46	https://zoom.us
4	https://youtube.com	47	https://adobe.com
6	https://microsoft.com	48	https://yandex.ru
7	https://twitter.com	49	https://vimeo.com
8	https://baidu.com	51	https://openai.com
9	https://cloudflare.com	53	https://gandi.net
10	https://instagram.com	57	https://taobao.com
11	https://netflix.com	63	https://bit.ly
12	https://apple.com	64	https://vk.com
13	https://linkedin.com	65	https://intuit.com
14	https://amazon.com	67	https://tiktok.com
15	https://bilibili.com	73	https://msn.com
18	https://wikipedia.org	74	https://mozilla.org
19	https://qq.com	76	https://weibo.com
20	https://live.com	79	https://blogspot.com
23	https://yahoo.com	81	https://spotify.com
28	https://bing.com	82	https://icloud.com
29	https://office.com	83	https://paypal.com
30	https://github.com	85	https://tumblr.com
31	https://reddit.com	87	https://nih.gov
32	https://pinterest.com	88	https://jd.com
33	https://wordpress.org	89	https://health.mil
34	https://whatsapp.com	92	https://skype.com
37	https://pg.com	93	https://canva.com

Tabulka B.1: Seznam padesátí webových stránek, na kterých bylo provedeno měření v kapitole 7.1. Seznam prošel ručním filtrováním a původně pochází ze studie Tranco [11].