

## **STRESZCZENIE**

Krótki wstęp.

Część główna, która powinna być nieco dłuższa. Całe streszczenie powinno zająć około pół strony.

Krótkie podsumowanie wniosków, wyników i ewentualnych proponowanych kolejnych kroków.

**Słowa kluczowe:** słowo kluczowe 1, słowo kluczowe 2, słowo kluczowe 3

**Dziedzina nauki i techniki, zgodnie z wymogami OECD:** elektrotechnika, elektronika, inżynieria informatyczna



## **ABSTRACT**

A short introduction.

The main part which should be a bit longer. The whole abstract should take approximately a half page.

A short summary of the outcomes, results, and proposed next steps (if any).

**Keywords:** keyword 1, keyword 2, keyword 3

**Field of science and technology in accordance with OECD requirements:** electrical engineering, electronic engineering, information engineering



## SPIS TREŚCI

<b>Podsumowanie wymagań formalnych pracy</b>	<b>3</b>
0.1. Wykorzystanie GenAI	3
0.2. Autorstwo rozdziałów i podrozdziałów	3
<b>Wykaz ważniejszych oznaczeń i skrótów</b>	<b>4</b>
<b>1. Wprowadzenie</b>	<b>5</b>
1.1. Cel pracy	5
1.2. Przegląd rozdziałów	5
<b>2. Istniejące rozwiązania</b>	<b>6</b>
2.1. Narzędzia do automatycznej konwersji C++ na Rust	6
2.2. Projekty open source migrujące z C++ do Rust	7
2.3. Migracja komponentu Stylo (CSS engine) z C++ do Rust w projekcie Firefox	7
2.3.1. Projekt Stylo	8
2.3.2. Przebieg migracji i integracja Stylo z Firefox	9
2.3.3. Rezultaty i znaczenie projektu Stylo	9
2.3.4. Sukces rynkowy Firefox Quantum	10
2.3.5. Wnioski projektu Stylo	10
2.4. Eksperymentalna przeglądarka Servo	10
2.4.1. Architektura i kluczowe komponenty Servo	10
2.4.2. Przebieg rozwoju i integracja Servo z Firefoxem	11
2.4.3. Rezultaty i znaczenie projektu Servo	11
2.4.4. Wnioski Servo	11
2.5. Środowisko wykonawcze Deno dla JavaScript, TypeScript i WebAssembly	11
2.5.1. Architektura i kluczowe komponenty Deno	11
2.5.2. Rezultaty i znaczenie projektu Deno	12
2.5.3. Wnioski Deno	12
2.6. Inicjatywy Mozilla wspierające migrację	12
<b>3. Migracja fragmentu kodu z języka programowania C++ na język Rust</b>	<b>13</b>
3.1. Wybór fragmentu kodu do migracji	13
3.2. Analiza biblioteki testcrasher	13
3.2.1. Cel biblioteki testcrasher	13
3.2.2. Architektura biblioteki testcrasher przed migracją	14
3.3. Strategia migracji i wykorzystane narzędzia	14
3.3.1. Strategia migracji biblioteki testcrasher	14
3.3.2. Wykorzystane narzędzia i technologie	15
<b>4. Testowanie i wnioski końcowe</b>	<b>16</b>
<b>Wykaz literatury</b>	<b>17</b>
<b>Wykaz rysunków</b>	<b>18</b>



## PODSUMOWANIE WYMAGAŃ FORMALNYCH PRACY

### 0.1. WYKORZYSTANIE GENAI

### 0.2. AUTORSTWO ROZDZIAŁÓW I PODROZDZIAŁÓW

**Tabela 1.** Autorstwo poszczególnych rozdziałów i podrozdziałów

rozdział	autor
1. Wprowadzenie	Dzmitry Hurynovich
2. Istniejące rozwiązania	Dzmitry Hurynovich
3. Migracja fragmentu kodu	Marcin Szmidt
4. Testowanie i wnioski końcowe	TBD

## **WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW**

CFD – Computational Fluid Dynamics (obliczeniowa mechanika płynów)

UE – Unia Europejska



## **1. WPROWADZENIE**

Tekst rozdziału do napisania w terminie późniejszym.

### **1.1. CEL PRACY**

Celem pracy dyplomowej jest analiza oraz praktyczna realizacja migracji fragmentu programu udostępnionego przez fundację Mozilla z języka C++ do języka Rust w celu oceny korzyści związanych z bezpieczeństwem i nowoczesnością kodu.

### **1.2. PRZEGLĄD ROZDZIAŁÓW**

W rozdziale drugim pracy przedstawiono istniejące rozwiązania dotyczące migracji kodu z języka C++ do języka Rust, ze szczególnym uwzględnieniem projektów realizowanych przez społeczność open source oraz inicjatyw fundacji Mozilla, które ilustrują praktyczne podejścia i narzędzia wspierające ten proces.

## 2. ISTNIEJĄCE ROZWIĄZANIA

W niniejszym rozdziale przedstawiono kilka przykładowych rozwiązań wykorzystywanych w procesach migracji kodu źródłowego z języka C++ do języka Rust, ze szczególnym uwzględnieniem narzędzi automatyzujących ten proces oraz doświadczeń z projektów open source, takich jak te rozwijane przez fundację Mozilla.

### 2.1. NARZĘDZIA DO AUTOMATYCZNEJ KONWERSJI C++ NA RUST

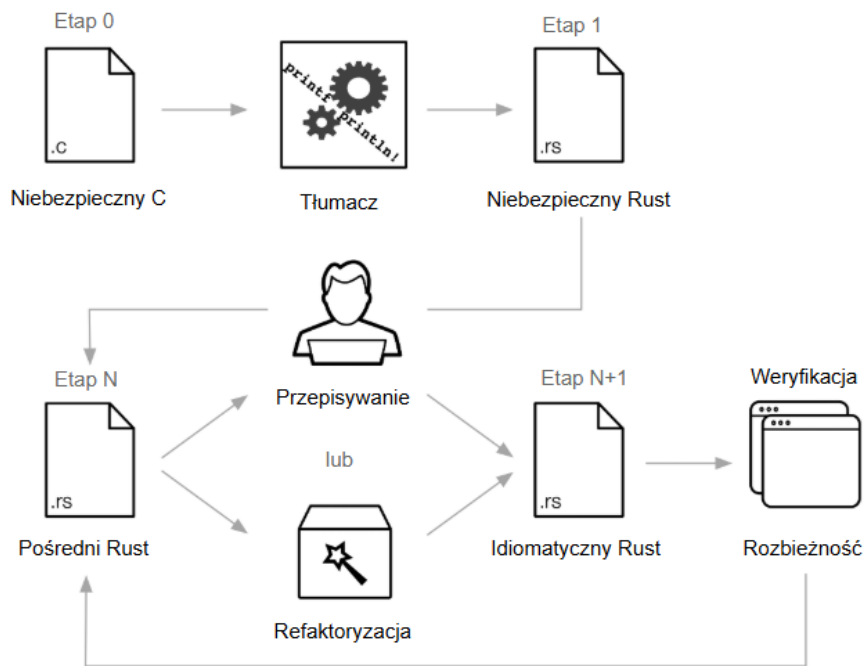
W procesie migracji kodu z C++ do Rust wykorzystywane są narzędzia wspomagające automatyzację, choć pełna konwersja nadal wymaga ręcznego dostosowania ze względu na różnice semantyczne między językami. Przykładowe narzędzia to:

- **C2Rust** - framework umożliwiający translację kodu C (i częściowo C++) do Rust, wykorzystujący Clang do parsowania kodu źródłowego. Narzędzie generuje niskopoziomowy kod, który wymaga późniejszej refaktoryzacji (np. wprowadzenia bezpiecznych abstrakcji). [1]
- **Bindgen** - narzędzie rozwijane przez Mozilla, automatycznie generujące powiązania (ang. bindings) kodu Rust do C/C++. [2]
- **Corrode** - eksperymentalny translator C do Rust.[3]

Narzędzia takie jak C2Rust generują kod w języku Rust, który jest oznaczony jako niebezpieczny (ang. unsafe). Nie oznacza to, że kod jest z natury wadliwy, ale że kompilator Rusta nie jest w stanie zweryfikować jego poprawności pod kątem bezpieczeństwa pamięci. W bloku unsafe programista zyskuje dostęp do pięciu dodatkowych operacji, niemożliwych w bezpiecznym Rustcie, takich jak dereferencja surowych wskaźników czy wywoływanie niebezpiecznych funkcji. W ten sposób programista przejmuje od kompilatora odpowiedzialność za zapewnienie, że operacje na pamięci są poprawne.[4]

Celem automatycznej translacji jest stworzenie działającego odpowiednika kodu C/C++, a nie wygenerowanie od razu bezpiecznego i idiomatycznego kodu Rust. Najlepszym podejściem jest stopniowe refaktoryzowanie kodu wygenerowanego przez translator, zastępując bloki unsafe bezpiecznymi abstrakcjami, aby w pełni wykorzystać gwarancje bezpieczeństwa, jakie oferuje Rust.[1]

Praca z tymi narzędziami może przebiegać w następujący sposób:



**Rys. 2.1.** Proces tłumaczenia i przekształcania kodu C na idiomatyczny kod w języku Rust(przygotowano na podstawie[1])

## 2.2. PROJEKTY OPEN SOURCE MIGRUJĄCE Z C++ DO RUST

- **Firefox (Mozilla)** - stopniowa migracja komponentów (np. silnik CSS Stylo), z wykorzystaniem Rust do poprawy bezpieczeństwa pamięci. Mozilla opracowała też RLBox, narzędzie do sandboxowania niebezpiecznego kodu C++.[5]
- **Servo** - eksperymentalna przeglądarka napisana całkowicie w Rust, której fragmenty (np. WebRender) zostały zintegrowane z Firefoxem.[6]
- **Deno** - (JavaScript lub TypeScript runtime) - używa Rust dla wydajnych modułów, podczas gdy core jest w C++.[7]
- **Linux Kernel** - od wersji 6.1 wspiera Rust jako drugi język systemowy, co umożliwia migrację wybranych modułów.[8]

Projekty te pokazują, że migracja często odbywa się modularnie, z zachowaniem interoperacyjności przez Foreign Function Interface(FFI).

## 2.3. MIGRACJA KOMPONENTU STYLO (CSS ENGINE) Z C++ DO RUST W PROJEKCIE FIREFOX

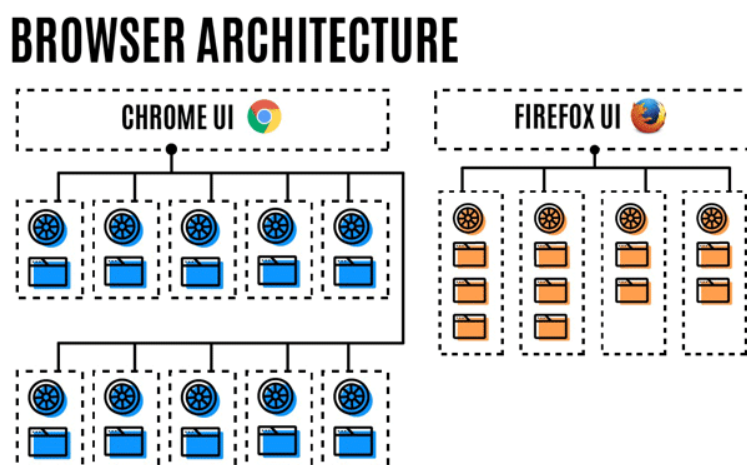
Przeglądarka Firefox, rozwijana przez fundację Mozilla, od wielu lat stanowi jedno z głównych środowisk testowych i produkcyjnych dla języka Rust. W ramach projektu *Quantum* zainicjowano serię modernizacji komponentów Firefox, której celem było poprawienie wydajności i bezpieczeństwa. Jednym z najbardziej znaczących efektów tej inicjatywy była migracja silnika CSS, znanego jako *Stylo* [9], z języka C++ do Rust. To przedsięwzięcie stanowi wzorcowy przypadek efektywnej migracji komponentu systemowego o wysokim stopniu złożoności.

### 2.3.1. PROJEKT STYLO

Silnik CSS odpowiada za przetwarzanie stylów arkuszy i ich stosowanie do drzewa DOM (Document Object Model) w czasie renderowania strony. Poprzedni silnik (*Gecko*), napisany w C++, miał ograniczoną możliwość wydajnej równoległości i był narażony na typowe problemy z zarządzaniem pamięcią. Rust, jako język systemowy bezpieczny pamięciowo, oferował realną szansę na poprawę niezawodności i skalowalności komponentu CSS[9].

Migracja Stylo została poprzedzona fazą eksperymentalną w ramach projektu Servo - nowej przeglądarki pisanej od podstaw w Rust. Na podstawie rezultatów z Servo, komponent *WebStylo* został przekształcony i zaadaptowany do Firefox jako *Stylo*.

Na rysunku 2.2 przedstawiono porównanie architektury wieloprocessowej w Chrome z hybrydowym podejściem zastosowanym w Firefox 57 (Quantum).



**Rys. 2.2.** Architektura Chrome w porównaniu do Firefox (na podstawie Firefox 57 "Quantum")  
[10]

Schemat pokazuje, że Chrome izoluje każdą kartę w osobnym procesie, podczas gdy Firefox Quantum grupuje karty w procesach treści, ograniczając tym samym zużycie pamięci.

#### **Klasyczna architektura wieloprocessowa (Chrome):**

- Wyspecjalizowane procesy: oddzielne dla każdej karty (tab), rozszerzeń (extensions) i GPU,
- Izolacja przez nadmiar: każda karta = nowy proces (wysokie zużycie RAM),
- Hierarchia kontrolna: proces główny (browser) zarządza procesami potomnymi.

#### **Podejście Quantum (Firefox):**

- Hybrydowy model procesów:
  - ★ jeden główny proces zarządzający (Parent),
  - ★ procesy treści (Content) współdzielone między kartami,
  - ★ dedykowane procesy dla krytycznych komponentów (GPU, Network),
- Optymalizacja zasobów:
  - ★ współdzielenie pamięci dla podobnych stron,
  - ★ dynamiczne alokowanie procesów wg potrzeb,

- Modułowość: wymienne komponenty i lepsza skalowalność.

#### Kluczowe komponenty nowej architektury:

- **Quantum Flow**

- ★ Priorytetyzacja zadań: System kolejek oparty o krytyczność operacji
- ★ Pipeline renderingu: Równoległe przetwarzanie etapów wyświetlania strony
- ★ Przeplot wątków: Wykorzystanie wszystkich rdzeni CPU

- **Quantum CSS**

- ★ Równoległe drzewo stylów: Podział pracy na niezależne fragmenty
- ★ Cache współdzielony: Jedna kopia stylów dla identycznych elementów
- ★ Inkrementalne aktualizacje: Minimalizacja przeróbek przy dynamicznych zmianach

- **Quantum Render (WebRender)**

- ★ Kompozytowanie na GPU: Traktowanie strony jako sceny 3D
- ★ Listy wyświetleń: Optymalizacja przekazywania danych do karty graficznej
- ★ Wektorowy pipeline: Bezstratne skalowanie elementów UI

#### 2.3.2. PRZEBIEG MIGRACJI I INTEGRACJA STYLO Z FIREFOX

Stylo został zaprojektowany jako komponent kompatybilny z istniejącym systemem budowania Firefoksa. Umożliwiło to tzw. *dual compilation* – kompilowanie części przeglądarki w Rust, a pozostałych w C++. Komunikacja między językami odbywa się poprzez FFI (Foreign Function Interface), co wymagało stworzenia bezpiecznych interfejsów i utrzymania zgodności ABI.

Migracja przebiegała etapami, zaczynając od funkcji odpowiedzialnych za selekcję stylów, a następnie przekształcając kolejne moduły[11]. Każdy etap podlegał intensywnemu testowaniu, zarówno funkcjonalnemu, jak i porównawczemu z poprzednią implementacją C++. Wprowadzenie Rust pozwoliło na równoległe przetwarzanie drzew stylów, co znacząco poprawiło wydajność renderowania.

#### 2.3.3. REZULTATY I ZNACZENIE PROJEKTU STYLO

Migracja silnika Stylo do Rust przyniosła korzyści:

- **Wydajność:** znaczący wzrost wydajności przeglądarki, szczególnie w obszarach dotyczących równoległego stylowania złożonych drzew DOM,
- **Bezpieczeństwo:** redukcja błędów pamięci typowych dla C++,
- **Inspiracja:** projekt stał się wzorem dla dalszych migracji komponentów Firefoksa.

Stylo jest wyłącznym rozwiązaniem Mozilla, rozwijanym tylko dla przeglądarek Servo i Firefox. Od wersji Firefox 57 (Quantum) zastąpił tradycyjny silnik Gecko CSS, wykorzystując architekturę zapoczątkowaną w projekcie Servo. Stylo działa jako hybrydowy silnik - w Firefox wykorzystuje zarówno komponenty Rust (Servo) jak i C++ (Gecko), podczas gdy w Servo istnieje jako czyste rozwiązanie w Rust.

- 80% redukcji błędów bezpieczeństwa pamięci

**Tabela 2.1.** Porównanie tradycyjnych silników CSS i Stylo (Quantum CSS)

Cecha	Gecko (C++)	Stylo (Rust)
Przebieg stylowania	Sekwencyjny	Równoległy
Bezpieczeństwo	Manualne zarządzanie pamięcią	Automatyczne (Rust)
Wydajność	1x	Do 18× na wielordzeniowych CPU
Kompatybilność	Wszystkie przeglądarki	Tylko Firefox/Servo

- 2-4× szybsze stylowanie stron
- 30% mniejsze zużycie RAM przy złożonych stylach

Projekt Stylo dowodzi, że migracja nawet bardzo złożonych komponentów systemowych jest możliwa i opłacalna, pod warunkiem dobrej integracji narzędzi, testów oraz wsparcia ze strony zespołu inżynierów. Stylo pozostaje jednym z flagowych przypadków użycia Rust w produkcyjnym środowisku i fundamentem sukcesu projektu Quantum.

#### 2.3.4. SUKCES RYNKOWY FIREFOX QUANTUM

Wprowadzenie silnika Stylo w Firefox 57 (*Quantum*) w listopadzie 2017 roku stanowiło punkt zwrotny dla przeglądarki Mozilli:

- Firefox odzyskał 15% użytkowników w ciągu 6 miesięcy od premiery,
- powstało ponad 100 nowych rozszerzeń stworzonych specjalnie dla Quantum,
- nagroda *WebAward* dla najszybszej przeglądarki 2018.

#### 2.3.5. WNIOSKI PROJEKTU STYLO

Przykład migracji Stylo pokazuje, że sukces transformacji kodu do Rust zależy nie tylko od możliwości technicznych, ale również od przyjętej strategii organizacyjnej i zdolności utrzymania kompatybilności z istniejącą bazą kodu. Mozilla, jako pionier wykorzystania Rust w praktyce, wyznaczyła kierunek rozwoju dla innych organizacji poszukujących nowoczesnych metod poprawy jakości oprogramowania systemowego.

### 2.4. EKSPERYMENTALNA PRZEGLĄDARKA SERVO

Servo to eksperymentalna przeglądarka internetowa rozwijana przez fundację Mozilla, napisana całkowicie w języku Rust. Głównym celem projektu było stworzenie nowoczesnego silnika przeglądarkowego, który wykorzystuje zalety Rust do poprawy wydajności i niezawodności. Servo stał się poligonem doświadczalnym dla wielu innowacyjnych rozwiązań, które później zintegrowano z Firefoxem[6].

#### 2.4.1. ARCHITEKTURA I KLUCZOWE KOMPONENTY SERVO

Servo został zaprojektowany z myślą o modularności i równoległym przetwarzaniu. Jego architektura obejmuje:

- Silnik renderowania *WebRender*: wykorzystuje GPU do komponowania stron, traktując je jako sceny 3D,
- Silnik stylów *Stylo*: równoległe przetwarzanie CSS, które później zostało włączone do Firefoxa,

- Parser HTML i DOM: zoptymalizowany pod kątem bezpieczeństwa i wydajności,
- Wsparcie dla WebAssembly: umożliwia wykonywanie wysokowydajnego kodu w przeglądarce.

#### 2.4.2. PRZEBIEG ROZWOJU I INTEGRACJA SERVO Z FIREFOXEM

Projekt Servo rozpoczął się w 2012 roku jako eksperyment mający na celu przetestowanie możliwości Rust w kontekście przeglądarki. W miarę rozwoju kluczowe komponenty Servo, takie jak *WebRender* i *Stylo*, zostały zintegrowane z Firefoxem w ramach projektu *Quantum*. Dzięki temu Firefox zyskał nowoczesne funkcje, zachowując jednocześnie kompatybilność z istniejącym kodem C++[5].

#### 2.4.3. REZULTATY I ZNACZENIE PROJEKTU SERVO

Servo przyniósł następujące korzyści:

- **Wydajność:** zastosowanie równoległego przetwarzania znacznie przyspieszyło renderowanie stron,
- **Bezpieczeństwo:** brak błędów pamięciowych (typowych dla C++),
- **Innowacje:** Servo stał się inspiracją dla innych projektów wykorzystujących Rust (np. Deno).

#### 2.4.4. WNIOSKI SERVO

Projekt Servo pokazał, że Rust nadaje się do budowy złożonych systemów, takich jak przeglądarki internetowe. Jego modularność i interoperacyjność z C++ umożliwiły stopniowe wdrażanie nowych rozwiązań w istniejących projektach, co jest kluczowe dla dużych organizacji.

### 2.5. ŚRODOWISKO WYKONAWCZE DENO DLA JAVASCRIPT, TYPESCRIPT I WEBASSEMBLY

Deno[7] to nowoczesne środowisko wykonawcze dla JavaScript, TypeScript i WebAssembly. Deno zostało napisane w Rust, co zapewnia mu wysoką wydajność i bezpieczeństwo. Głównym celem projektu było rozwiązanie problemów Node.js, takich jak złożony system zarządzania zależnościami i brak wsparcia dla TypeScript out-of-the-box.

#### 2.5.1. ARCHITEKTURA I KLUCZOWE KOMPONENTY DENO

Deno opiera się na następujących komponentach:

- Rust jako podstawa: większość funkcji systemowych jest zaimplementowana w Rust,
- Modułowy system bezpieczeństwa: Deno domyślnie uruchamia kod w sandboxie, co minimalizuje ryzyko zagrożeń,
- Wsparcie dla WebAssembly: umożliwia wykonywanie kodu napisanego w innych językach,
- Silnik V8: ten sam silnik JavaScript używany w Chrome i Node.js.

Deno wykorzystuje język Rust do implementacji niskopoziomowych funkcji, takich jak operacje wejścia/wyjścia (I/O) czy zarządzanie procesami. Komunikacja między JavaScriptem a Rust odbywa się za pośrednictwem interfejsu Foreign Function Interface (FFI), co umożliwia zachowanie wysokiej wydajności przy jednoczesnym zapewnieniu bezpieczeństwa.

### 2.5.2. REZULTATY I ZNACZENIE PROJEKTU DENO

Deno przyniósł następujące korzyści:

- **Wydajność:** dzięki Rust Deno osiąga lepszą wydajność niż Node.js (w niektórych zadaniach),
- **Bezpieczeństwo:** sandboxing i domyślne ograniczenia minimalizują ryzyko ataków,
- **Nowoczesne funkcje:** wsparcie dla TypeScript i WebAssembly out-of-the-box.

### 2.5.3. WNIOSKI DENO

Deno jest przykładem udanego połączenia JavaScript i Rust, pokazując, że migracja wybranych komponentów do Rust może przynieść znaczące korzyści w zakresie wydajności i bezpieczeństwa.

## 2.6. INICJATYWY MOZILLA WSPIERAJĄCE MIGRACJĘ

Mozilla, jako jeden z głównych fundatorów rozwoju Rust, prowadzi projekty ułatwiające przejście z C++:

- **Oxidization** - wewnętrzny program Mozilla mający na celu identyfikację komponentów Firefox, których migracja do Rust przyniesie największe korzyści bezpieczeństwa[12],
- **CXX** - biblioteka do bezpiecznej interoperacyjności C++ i Rust, minimalizująca ryzyko błędów na styku języków[13],
- **Rust-C++ dual compilation** - wsparcie w build systemie Firefox dla mieszanych projektów[14].

Działania te pokazują, że migracja w dużych projektach wymaga nie tylko narzędzi, ale też wsparcia organizacyjnego i rozwoju oprogramowania.



### 3. MIGRACJA FRAGMENTU KODU Z JĘZYKA PROGRAMOWANIA C++ NA JĘZYK RUST

#### 3.1. WYBÓR FRAGMENTU KODU DO MIGRACJI

Podczas wyboru fragmentu kodu kierowaliśmy się kilkoma kluczowymi kryteriami:

- **Wysoki potencjał poprawy bezpieczeństwa:**
  - ★ **Zarządzanie pamięcią:** Kandydat do migracji powinien operować w obszarze, w którym błędy zarządzania pamięcią, typowe dla języka C++, mogą prowadzić do poważnych luk w zabezpieczeniach. Przykładem może być praca z surowymi danymi, takimi jak zrzuty pamięci.
  - ★ **Zastąpienie przestarzałej zależności:** Komponent opiera się na zewnętrznej bibliotece C++, uznawanej za przestarzałą lub posiadającej nowocześniejszy i bezpieczniejszy odpowiednik w języku Rust.
- **Wykonalność i modularność:** Fragment kodu musiał być na tyle odizolowany, aby jego migracja nie pociągała za sobą konieczności przepisywania znacznych części przeglądarki. Biblioteka o jasno zdefiniowanym API i konkretnym zadaniu ułatwia proces zastępowania implementacji bez naruszania reszty systemu.
- **Zgodność ze strategicznymi celami Fundacji Mozilla:** Wybrany komponent powinien wpisywać się w długofalową strategię Mozilli polegającą na stopniowym zwiększaniu ilości kodu napisanego w Rust w celu poprawy bezpieczeństwa i wydajności przeglądarki. Zgodnie z inicjatywą "Oxidation". [12]

Na podstawie tych cech do migracji została wybrana wewnętrzna biblioteka testcrasher.dll. Na decyzję dodatkowo wpłynął fakt istnienia w systemie śledzenia błędów Mozilli zadania o numerze Bug 1798688[15], które jawnie definiuje cel jako "Replace breakpad with rust-minidump in the testcrasher library"<sup>1</sup>.

#### 3.2. ANALIZA BIBLIOTEKI TESTCRASHER

##### 3.2.1. CEL BIBLIOTEKI TESTCRASHER

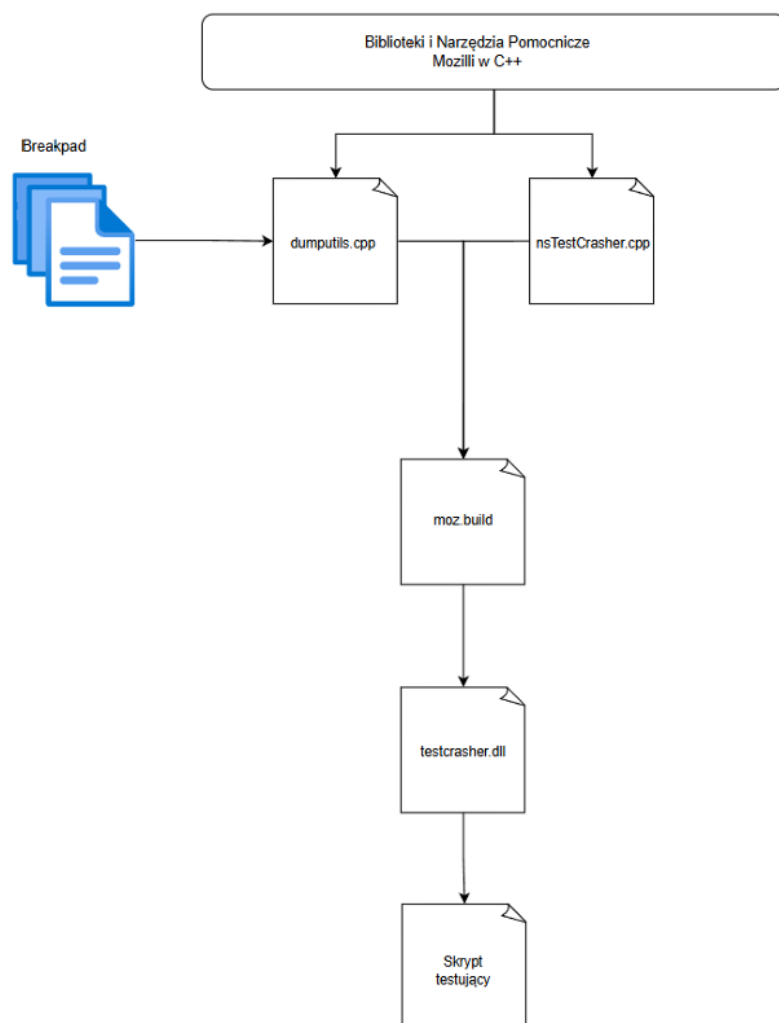
Biblioteka dynamiczna testcrasher jest używana jako tester działania komponentu crashreporter w Mozilla Firefox. Jej działanie skupia się na dwóch obszarach:

- **Analiza zrzutu pamięci:** Za tą część odpowiada plik dumputils.cpp.
- **Wywoływanie awarii procesu:** Za tą część odpowiada plik nsTestCrasher.cpp.

---

<sup>1</sup>Zastąpienie breakpad przez rust-minidump w bibliotece testcrasher

### 3.2.2. ARCHITEKTURA BIBLIOTEKI TESTCRASHER PRZED MIGRACJĄ



Rys. 3.1. architektura biblioteki testcrasher

## 3.3. STRATEGIA MIGRACJI I WYKORZYSTANE NARZĘDZIA

### 3.3.1. STRATEGIA MIGRACJI BIBLIOTEKI TESTCRASHER

Przyjęta strategia migracji opiera się na stopniowym i iteracyjnym zastępowaniu kodu C++ kodem Rust, przy jednoczesnym zachowaniu w pełni kompatybilnego publicznego API. Fundamentalnym założeniem jest, że z perspektywy klienta biblioteki, którym w tym przypadku są skrypty testujące, proces migracji jest całkowicie transparentny. Wymaga to utrzymania stabilnego interfejsu binarnego aplikacji (ABI) zgodnego z językiem C. Dzięki temu poszczególne funkcje, a docelowo całe moduły zaimplementowane w C++, mogą być zastępowane ich odpowiednikami w Rust, a następnie weryfikowane za pomocą istniejącego zestawu testów. Proces migracji został zaplanowany w następujących, logicznie następujących po sobie etapach:

1. **Konfiguracja hybrydowego procesu budowania:** Pierwszym krokiem jest modyfikacja systemu budowania Mozilli (moz.build) w celu umożliwienia współlistnienia kodu C++ i Rust.

Polega to na zdefiniowaniu reguł kompilacji dla nowego kodu Rust do postaci biblioteki statycznej (`rust_testcrasher.lib`). Następnie, ta biblioteka statyczna jest dołączana do finalnej biblioteki dynamicznej (`testcrasher.dll`), a jej publiczne symbole są eksportowane w taki sposób, aby zachować zgodność z oryginalnym API.

2. **Iteracyjna migracja logiki analitycznej (`dumputils.cpp`):** Proces właściwej migracji rozpoczyna się od komponentów analitycznych. Poszczególne funkcje odpowiedzialne za parsowanie i analizę plików minidump są reimplementowane w języku Rust, wykorzystując do tego celu bibliotekę `'rust-minidump'`. Po zaimplementowaniu każdej funkcji w Rust, jej oryginalna wersja w C++ jest usuwana, a nowa implementacja zostaje zintegrowana w procesie budowania.
3. **Migracja logiki inicjującej awarie (`nsTestCrasher.cpp`):** Po pomyślnej weryfikacji poprawności działania modułu analitycznego, analogiczny proces jest stosowany do kodu odpowiedzialnego za inicjowanie stanów awaryjnych. Funkcje C++ są zastępowane przez ich odpowiedniki w Rust, które wykorzystują bibliotekę `'sadness-generator'`.
4. **Finalizacja i czyszczenie konfiguracji:** Po zakończeniu migracji całości kodu funkcjonalnego do języka Rust, oryginalne pliki źródłowe C++ (`'dumputils.cpp'` oraz `'nsTestCrasher.cpp'`) są ostatecznie usuwane z drzewa projektu. Konfiguracja w pliku `'moz.build'` jest upraszczana, eliminując reguły dotyczące kompilacji C++ dla tej biblioteki.

Dzięki takiemu podejściu, w dowolnym momencie procesu migracji biblioteka pozostaje w pełni funkcjonalna, zawierając mieszankę działającego kodu C++ i Rust.

### 3.3.2. WYKORZYSTANE NARZĘDZIA I TECHNOLOGIE

Do przeprowadzenia migracji wybranego fragmentu kodu użyto:

-

#### **4. TESTOWANIE I WNIOSKI KOŃCOWE**

Tekst do napisania w terminie późniejszym.

## WYKAZ LITERATURY

- [1] Galois, Inc. and Immunant, Inc. *C2Rust Manual*. <https://c2rust.com/manual>. 2023.
- [2] The Rust Programming Language. *The bindgen User Guide*. <https://rust-lang.github.io/rust-bindgen>. 2025.
- [3] J. Sharp. *Corrode: C to Rust Translator*. <https://github.com/jameysharp/corrode>. 2021.
- [4] rust-lang. *Unsafe Rust - The Rust Programming Language - Rust Documentation*. <https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html>. 2025.
- [5] Mozilla Foundation. *Quantum/Stylo Project*. <https://wiki.mozilla.org/Quantum/Stylo>. 2017.
- [6] Servo Project. *Servo*. <https://github.com/servo/servo>. 2025.
- [7] Deno Team. *Deno Runtime Documentation*. <https://docs.deno.com/runtime/>. 2024.
- [8] Linux Kernel Developers. *Rust in the Linux Kernel – Quick Start*. <https://docs.kernel.org/rust/quick-start.html>. 2024.
- [9] L. Clark. *Inside a super fast CSS engine: Quantum CSS (aka Stylo)*. <https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo>. 2017.
- [10] R. Ayeshani. *How web browsers use process & Threads ? (Firefox VS Chrome)*. <https://randiayeshani.medium.com/how-web-browsers-use-process-threads-305b5a2164d4>. 2020.
- [11] Wikipedia contributors. *Quantum/Stylo*. <https://wiki.mozilla.org/Quantum/Stylo>. 2018.
- [12] Mozilla Foundation. *Oxidation Initiative*. <https://wiki.mozilla.org/Oxidation>. 2020.
- [13] CXX Project. *CXX: Safe FFI between Rust and C++*. <https://cxx.rs/>. 2024.
- [14] Mozilla Build Team. *Rust in Firefox Build System*. <https://firefox-source-docs.mozilla.org/build/buildsystem/rust.html>. 2024.
- [15] G. Svelto. *Replace breakpad with rust-minidump in the testcrasher library*. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1798688](https://bugzilla.mozilla.org/show_bug.cgi?id=1798688). 2022.

## WYKAZ RYSUNKÓW

2.1. Proces tłumaczenia i przekształcania kodu C na idiomatyczny kod w języku Rust(przygotowano na podstawie[1]) . . . . .	7
2.2. Architektura Chrome w porównaniu do Firefox (na podstawie Firefox 57 "Quantum")	8
3.1. architektura biblioteki testcrasher . . . . .	14

## WYKAZ TABEL

1. Autorstwo poszczególnych rozdziałów i podrozdziałów . . . . .	3
2.1. Porównanie tradycyjnych silników CSS i Stylo (Quantum CSS) . . . . .	10