POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko studenta: Marcin Szmidt
Nr albumu: 188766
Poziom kształcenia: studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Profil: Teleinformatyka

Imię i nazwisko studenta: Dzmitry Hurynovich
Nr albumu: 191529
Poziom kształcenia: studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Profil: Systemy geoinformatyczne

PRACA DYPLOMOWA INŻYNIERSKA

Tytuł pracy w języku polskim: Migracja z języka programowania C++ na język Rust fragmentu programu udostępnionego przez fundację Mozilla

Tytuł pracy w języku angielskim: Migration of a part of an application released by the Mozilla foundation from C++ to the Rust programming language

Opiekun pracy: dr inż. Adam Kaczmarek

STRESZCZENIE

W niniejszej pracy znajduje się opis procesu migracji kodu należącego do fundacji Mozilla z języka C++ na język Rust. Przedmiotem praktycznej realizacji jest biblioteka dynamiczna testcrasher, wykorzystywana w procesie testowania mechanizmów raportowania błędów w przeglądarce Firefox. Głównym celem pracy jest wykonanie takiej migracji oraz ocena korzyści związanych z bezpieczeństwem pamięci i nowoczesnością kodu, jakie oferuje język Rust. Oznacza to porównanie kodu przed i po migracji pod kątem zarządzania pamięcią, szybkością działania oraz utrzymania oprogramowania. Przedstawione zostają kolejne kroki podjęte w celu przeprowadzenia migracji. Najpierw wyznaczono kryteria według których dokonano wyboru fragmentu kodu, kolejne sekcje obejmują jego szczegółowa analiza i jego stopniowe przepisanie na język Rust. Wyniki prac potwierdziły, że możliwa jest stopniowa wymiana kodu C++ na Rust w dużym projekcie przy zachowaniu pełnej kompatybilności funkcjonalnej. Nowa implementacja eliminuje ryzyko typowych błędów zarządzania pamięcią dzięki wykorzystaniu modelu własności języka Rust, co zweryfikowano za pomocą testów regresji.

Słowa kluczowe: C++, Rust, migracja kodu, bezpieczeństwo pamięci, Mozilla Firefox

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

ABSTRACT

This thesis describes the process of migrating code belonging to the Mozilla Foundation from C++ to Rust. The subject of the practical implementation is the testcrasher dynamic library, used for testing error reporting mechanisms in the Firefox browser. The main objective of this thesis is to execute this migration and evaluate the benefits regarding memory safety and code modernity offered by the Rust language. This involves comparing the code before and after the migration in terms of memory management, performance, and software maintainability. The subsequent steps taken to conduct the migration are presented. First, the criteria used to select the code fragment were defined; subsequent sections cover its detailed analysis and gradual rewriting into Rust. The results confirmed that a gradual replacement of C++ code with Rust in a large-scale project is possible while maintaining full functional compatibility. The new implementation eliminates the risk of typical memory management errors by utilizing Rust's ownership model, which was verified using regression tests.

Keywords: C++, Rust, code migration, memory safety, Mozilla Firefox

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

SPIS TREŚCI

PODSUMOWANIE WYMAGAŃ FORMALNYCH PRACY

0.1. WYKORZYSTANIE GENAI

Oświadczam, iż praca została wytworzona samodzielnie i bez wykorzystania narzędzi GenAI

0.2. AUTORSTWO ROZDZIAŁÓW I PODROZDZIAŁÓW

Tabela 1. Autorstwo poszczególnych rozdziałów i podrozdziałów

rozdział	**autor**
Streszczenie	Marcin Szmidt
1. Wprowadzenie	Marcin Szmidt
2. Istniejące rozwiązania	Dzmitry Hurynovich
3-3.5.1 Migracja fragmentu kodu	Marcin Szmidt
3.5.2 Migracja fragmentu kodu	Dzmitry Hurynovich
4. Testowanie i wnioski końcowe	Dzmitry Hurynovich
5. Podsumowanie	Dzmitry Hurynovich

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

ABI	–	Application Binary Interface
API	–	Application Programming Interface
CPU	–	Central Processing Unit
CSS	–	Cascading Style Sheets
DLL	–	Dynamic Link Library
DOM	–	Document Object Model
FFI	–	Foreign Function Interface
GPU	–	Graphics Processing Unit
I/O	–	Input/Output
MSVC	–	Microsoft Visual C++

1. WPROWADZENIE

Język C++ od dziesięcioleci stanowi fundament systemów operacyjnych, silników gier oraz przeglądarek internetowych, w tym projektu Mozilla Firefox. Nadal pozostaje jednym z kluczowych języków w branży IT. Według raportu Stack Overflow Developer Survey 2025, 23.5% programistów na świecie wykorzystuje ten język w swojej codziennej pracy[1]. Jego główną zaletą jest wysoka wydajność i kontrola nad sprzętem. Wiąże się to jednak z koniecznością ręcznego zarządzania pamięcią, co w skali milionów linii kodu prowadzi do trudnych do wykrycia błędów. Statystyki gigantów technologicznych, takich jak Microsoft czy Google, wskazują, że około 70% wszystkich krytycznych luk bezpieczeństwa w ich oprogramowaniu systemowym wynika właśnie z błędów zarządzania pamięcią[2].

W kontekście przeglądarek internetowych problem ten jest szczególnie istotny. Przetwarzają one niezaufane dane pochodzące z sieci, przez co często stają się wektorem ataku. Błędy takie jak przepełnienie bufora czy użycie zwolnionej pamięci mogą prowadzić nie tylko do awarii programu, ale dodatkowo umożliwiają atakującym wykonanie dowolnego kodu oraz kradzież wrażliwych informacji. Pomimo że C++ zapewnia programiście dużą swobodę i wydajność, odbywa się to kosztem braku wbudowanych mechanizmów weryfikacji odwołań do pamięci. Mimo istnienia nowoczesnych standardów C++ oraz narzędzi do analizy statycznej i dynamicznej, eliminacja tej klasy błędów w tak złożonych projektach pozostaje wyzwaniem generującym wysoki dług technologiczny. Z tego powodu organizacje takie jak NSA rekomendują odejście od języków wymagających ręcznego zarządzania pamięcią na rzecz języków bezpiecznych pamięciowo, do których zalicza się m.in. Rust.[3]

Jednym z rozwiązań problemów związanych z długiem technologicznym i bezpieczeństwem pamięci jest język programowania Rust. Jedną z jego kluczowych cech była rezygnacja z mechanizmu Garbage Collection na rzecz unikalnego modelu własności (ownership), pożyczania (borrowing) i „lifetimes", by zapewnić bezpieczeństwo pamięci przy wydajności zbliżonej do C/C++. Model własności to zestaw zasad które określają sposób w jaki program napisany w języku Rust zarządza pamięcią. Dodatkowo bezpieczeństwo kodu zwiększane jest dzięki zapewnieniu bezpieczeństwa typów - użycie `Result<T,E>` wymusza na programiście napisanie obsługi dla błędów dzięki czemu kod staje się bardziej przewidywalny. Jednocześnie użycie Rusta wiąże się z pewnymi ograniczeniami. Kompilator wykonuje rozbudowaną analizę borrow-checkera i optymalizacje, co powoduje wydłużenie czasu kompilacji, szczególnie przy dużych projektach. Ponadto migracja istniejącego kodu C++ do Rust lub współpraca obu języków wymaga dodatkowych warstw pośrednich, uwzględnienia różnic w konwencjach pamięci i ABI oraz starannego projektowania interfejsów, co czyni proces integracji bardziej złożonym.

Wprowadzenie nowego języka do istniejącego, ogromnego projektu, jakim jest Firefox, ze względów praktycznych nie może odbyć się poprzez całkowite przepisanie kodu. Baza kodu C++ jest zbyt duża i zbyt cenna, aby ją porzucić. Z tego powodu najczęstszym podejściem jest migracja stopniowa, polegająca na wymianie poszczególnych modułów i łączeniu ich za pomocą interfejsów binarnych. Niniejsza praca skupia się na praktycznym aspekcie tego procesu na przykładzie wybranego komponentu.

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. Praca stanowi Proof of

Concept, mający na celu wykazanie, że możliwe jest zastąpienie kodu typu legacy bezpiecznym i nowoczesnym kodem w języku Rust, przy zachowaniu pełnej zgodności funkcjonalnej i binarnej.

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.

W rozdziale trzecim szczegółowo przedstawiono praktyczny proces migracji. Opis rozpoczyna się od omówienia kryteriów, które doprowadziły do wyboru migrowanego fragmentu kodu, a następnie prezentuje jego dogłębną analizę, przyjętą strategię działania, użyte narzędzia oraz finalny, wieloetapowy proces zastępowania kodu C++ kodem Rust.

W rozdziale czwartym podjęto się analizy i testów uzyskanych rezultatów oraz przedstawiono techniczne aspekty pracy. Rozpoczęto od omówienia środowiska testowego i opisano przyjętą metodologię. W kolejnych sekcjach przeprowadzono testy funkcjonalne, których celem było potwierdzenie poprawnego działania kodu. Porównano ze sobą również implementacje w C++ i Rust, biorąc pod uwagę między innymi rozmiar pliku wynikowego oraz czas kompilacji. Rozdział zakończono opisem problemów napotkanych w trakcie przeprowadzania migracji.

W rozdziale piątym przedstawiono podsumowanie wykonanych prac oraz sformułowano wnioski końcowe dotyczące korzyści płynących z zastosowania języka Rust. Wskazano również potencjalne kierunki dalszego rozwoju projektu.

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). [4]

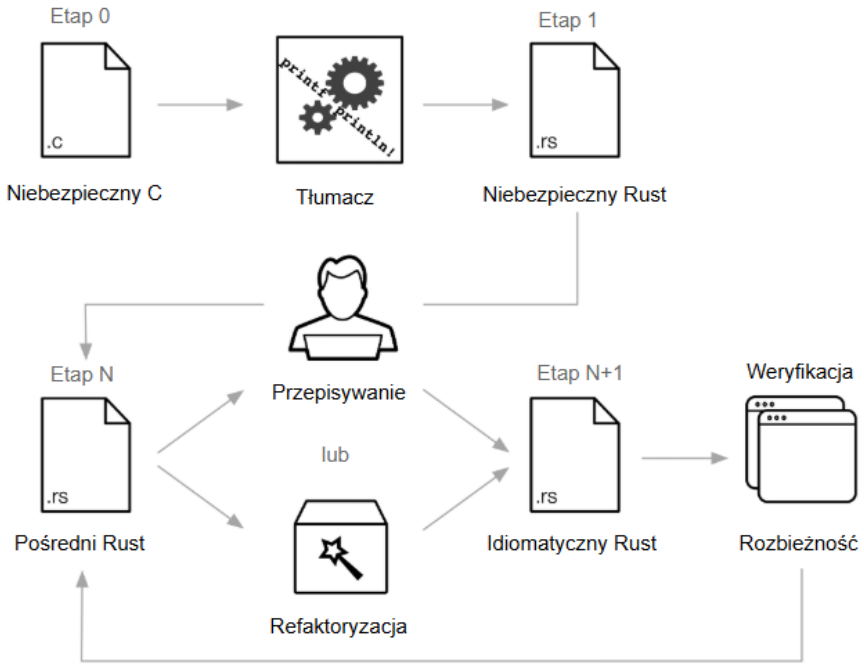
- **Bindgen** - narzędzie rozwijane przez Mozilla, automatycznie generujące powiązania (ang. bindings) kodu Rust do C/C++. [5]

- **Corrode** - eksperymentalny translator C do Rust.[6]

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 Ruście, 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.[7]

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.[4]

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[4])

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++.[8]

- **Servo** - eksperymentalna przeglądarka napisana całkowicie w Rust, której fragmenty (np. WebRender) zostały zintegrowane z Firefoxem.[9]

- **Deno** - (JavaScript lub TypeScript runtime) - używa Rust dla wydajnych modułów, podczas gdy core jest w C++.[10]

- **Linux Kernel** - od wersji 6.1 wspiera Rust jako drugi język systemowy, co umożliwia migrację wybranych modułów.[11]

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

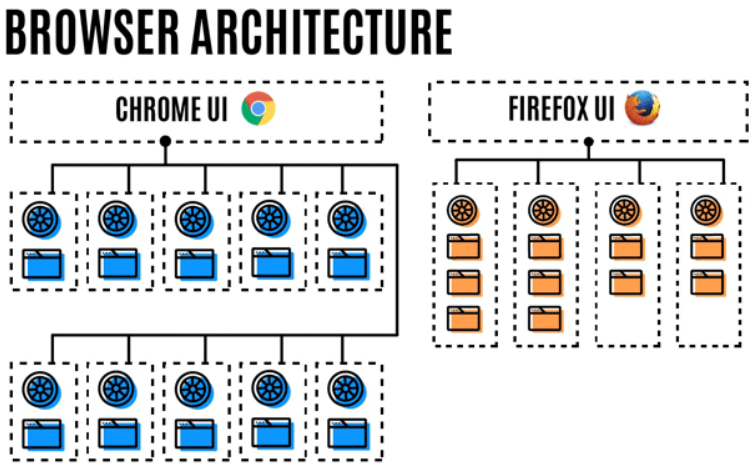
Przeglądarka Firefox, rozwijana przez fundację Mozilla, od wielu lat stanowi jedno z głównych środowisk testowych i produkcyjnych dla języka Rust. W ramach projektu *Quantum* zainicjowano serię modernizacji komponentów Firefox, której celem było poprawienie wydajności i bezpieczeństwa. Jednym z najbardziej znaczących efektów tej inicjatywy była migracja silnika CSS, znanego jako *Stylo* [12], 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[12].

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 wieloprocesowej w Chrome z hybrydowym podejściem zastosowanym w Firefox 57 (Quantum).



Rys. 2.2. Architektura Chrome w porównaniu do Firefox (na podstawie Firefox 57 "Quantum") [13]

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 wieloprocesowa (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[14]. 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[9].

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++[8].

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[10] 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[15],
- **CXX** - biblioteka do bezpiecznej interoperacyjności C++ i Rust, minimalizująca ryzyko błędów na styku języków[16],
- **Rust-C++ dual compilation** - wsparcie w build systemie Firefox dla mieszanych projektów[17].

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

Niniejszy rozdział szczegółowo opisuje proces migracji wybranego komponentu z języka programowania C++ na język Rust. Zgodnie z założeniami projektu, poszukiwania odpowiedniego fragmentu kodu ograniczono do oprogramowania rozwijanego przez Fundację Mozilla, co w praktyce skierowało uwagę na bazę kodu przeglądarki Firefox. Główną motywacją dla podjętych działań jest dążenie do poprawy bezpieczeństwa pamięci i ogólnej stabilności aplikacji, co jest jednym ze strategicznych celów wykorzystania języka Rust w dojrzałych projektach. W dalszej części rozdziału przedstawiono kolejne etapy pracy: począwszy od kryteriów, które zadecydowały o wyborze komponentu, przez jego szczegółową analizę, aż po opis przyjętej strategii migracji, wykorzystanych narzędzi i finalnego przebiegu implementacji.

3.1. KRYTERIA WYBORU FRAGMETU KODU DO MIGRACJI

Podczas wyboru fragmentu kodu kierowaliśmy się kilkoma kluczowymi kryteriami:

- **Wysoki potencjał poprawy bezpieczeństwa:**
  - ⋆ **Zarządznie 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 incjatywą "Oxidation". [15]

3.2. WYBRANY FRAGMENT KODU - BIBLIOTEKA DYNAMICZNA TESTCRASHER

Pierwszym etapem prac było zidentyfikowanie w kodzie źródłowym należącym do fundacji Mozilla odpowiedniego kandydata do migracji, który spełniałby wcześniej zdefiniowane kryteria. W procesie tym wykorzystano dwa narzędzia. Pierwszym z nich był Searchfox, narzędzie które indeksuje kod źródłowy oraz umożliwia szybkie wyszukiwanie kodu i plików źródłowych. Drugim narzędziem była Bugzilla, system śledzenia zadań Mozilli, służący do zarządzania zgłoszeniami błędów, propozycjami zmian i zadaniami deweloperskimi. Przy użyciu tych narzędzi oraz na podstawie powyżej opisanych kryteriów do migracji została wybrana wewnętrzna biblioteka testcrasher.dll. Na decyzję dodatkowo wpłynął fakt istnienia w systemie śledzenia błędów Mozilli zadania o numerze Bug 1798688[18], które jawnie definiuje cel jako "Replace breakpad with

rust-minidump in the testcrasher library"[1].

3.2.1. CEL BIBLIOTEKI TESTCRASHER

Biblioteka dynamiczna testcrasher jest narzędziem deweloperskim które używane jest jako tester działania komponentu Crashreporter - wewnętrznego mechanizmu przeglądarki Firefox, odpowiedzialnego za zbieranie i raportowanie informacji o awariach aplikacji. Działa poprzez wywoływanie awarii procesu a następnie analizę pliku zrzutu pamięci (.dmp) który został wytworzony przez moduł Crashreporter. Jej działanie skupia się na dwóch obszarach:

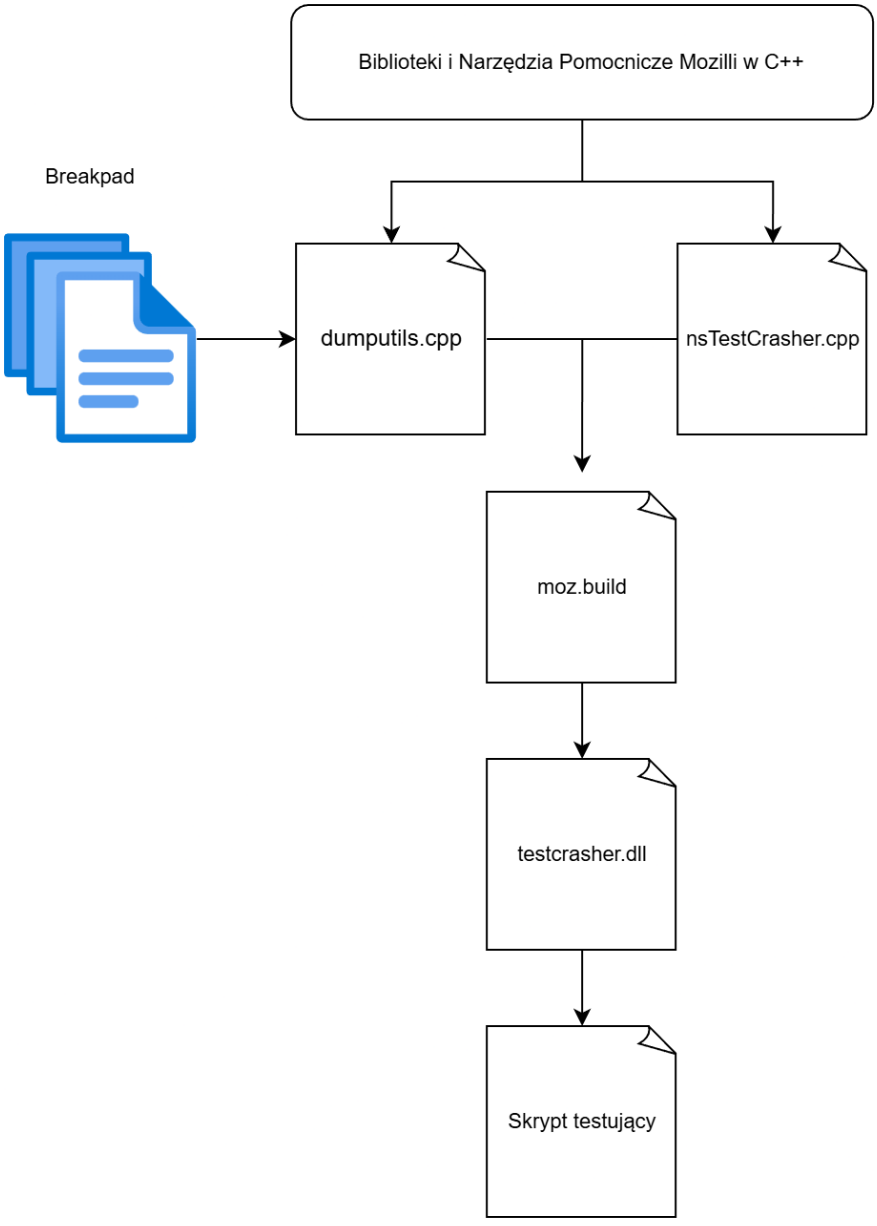
- **Analiza zrzutu pamięci:** Za tą część odpowiada plik dumputils.cpp. API tej części zawiera:
  - ⋆ **DumpHasStream()** - Zwraca wartość `true`, jeśli dany zrzut pamięci zawiera strumień określonego typu.
  - ⋆ **DumpHasInstructionPointerMemory()** - Zwraca wartość `true`, jeśli dany zrzut pamięci zawiera region pamięci który zawiera wskaźnik instrukcji z rekordu wyjątku.
  - ⋆ **DumpCheckMemory()** - Sprawdza, czy zrzut pamięci zawiera region rozpoczynający się pod adresem określonym w pliku `crash-addr` w bieżącym katalogu roboczym. Region ten musi mieć długość 32 bajtów i zawierać wartości od 0 do 31 w porządku rosnącym.
- **Wywoływanie awarii procesu:** Za tą część odpowiada plik nsTestCrasher.cpp. API tej części zawiera:
  - ⋆ **Crash():** Funkcja pełniąca rolę głównego dyspozytora, odpowiedzialna za celowe wywołanie awarii procesu. Na podstawie przekazanego argumentu uruchamia odpowiednią implementację błędu.
  - ⋆ **EnablePHC():** Aktywuje mechanizm Probabilistic Heap Checker.
  - ⋆ **GetWin64CFITestFnAddrOffset():** Oblicza i zwraca relatywny adres wirtualny (RVA) funkcji asemblerowych służących do testów CFI (Control Flow Integrity), w odniesieniu do adresu bazowego biblioteki testcrasher.dll.
  - ⋆ **TryOverrideExceptionHandler():** Rejestruje niestandardowy filtr wyjątków nieobsłużonych (poprzez API Windows `SetUnhandledExceptionFilter`), który wymusza natychmiastowe zakończenie procesu w momencie wystąpienia błędu.
  - ⋆ **SaveAppMemory():** Alokuje testowy bufor pamięci, wypełnia go określonym wzorcem danych, a następnie zapisuje jego adres do pliku `crash-addr`. Służy to weryfikacji, czy mechanizm raportowania awarii poprawnie dołącza wskazane, dodatkowe obszary pamięci aplikacji do wygenerowanego pliku zrzutu (minidump).

3.2.2. ARCHITEKTURA BIBLIOTEKI TESTCRASHER PRZED MIGRACJĄ

Rdzeniem biblioteki dynamicznej testcrasher są dwa pliki źródłowe C++: dumputils.cpp i nsTestCrasher.cpp. Moduł dumputils.cpp wykorzystuje do swojego działania zewnętrzną bibliotekę Google Breakpad. Jest to projekt open-source napisany w C++, dostarczający API do obsługi plików minidump. Jednym z głównych zadań tej pracy jest pozbycie się tej zależności.
Kod źródłowy biblioteki korzysta w niewielkim stopniu z kluczowych mechanizmów i konwencji specyficznych dla bazy kodu Firefoksa. Powoduje to że migracja kodu z Języka C++ na Język Rust nie będzie wiązać się z przepisywaniem/modyfikacjami kodu poza biblioteką testcrasher.

[1] Zastąpienie breakpad przez rust-minidump w blibliotece testcrasher

System budowania Mozilli, w przypadku biblioteki testcrasher stosuje metodę gdzie pliki .cpp są łączone w jedną jednostkę kompilacji w celu przyspieszenia procesu. Następnie skompilowany kod obiektowy jest linkowany w ostateczną bibliotekę dynamiczną testcrasher.dll.
Głównym sposobem użycia biblioteki sa zautomatyzowane testy, najczęściej pisane w JavaScripcie i uruchamiane w specjalnym frameworku testowym Mozilli. Skrypt testowy wywołuje wyeksportowaną funkcję z testcrasher.dll w celu spowodowania awarii i sprawdzenia czy moduł Crash Reporter wygenerował poprawny plik zrzutu pamięci.



Rys. 3.1. Architektura biblioteki testcrasher - opracowanie własne

3.3. STRATEGIA MIGRACJI I WYKORZYSTANE NARZĘDZIA

Po wyborze biblioteki testcrasher jako fragmentu kodu do migracji na język Rust kolejnym ważnym aspektem był dobór odpowiedniej strategii oraz narzędzi. Obrane podejście musiało uwzględniać specyfikę pracy z dużą bazę kodu jaką, jest projekt Mozilla Firefox. Priorytetem stało się zatem zapewnienie bezpieczeństwa samego procesu migracji, zdefiniowanego jako brak regresji -

istniejące i działające funkcjonalności nie zostaną uszkodzone przez wprowadzane zmiany podczas procesu migracji oraz zachowanie stabilności binarnej (ABI zgodne z językiem C). Aby to osiągnąć, każda nowo napisana w Rust funkcja była natychmiast integrowana i weryfikowana za pomocą istniejącego zestawu testów automatycznych, co gwarantowało jej pełną kompatybilność z resztą systemu.

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 niewidoczny. 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 procesu budowania:** Pierwszym krokiem jest modyfikacja systemu budowania Mozilli (moz.build) w celu umożliwienia współistnienia 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 crate 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. WYKORZYTANE NARZĘDZIA I TECHNOLOGIE

Do przeprowadzenia migracji wybranego fragmentu kodu użyto następujących narzędzi i technologi:

- **Języki programowania i platforma:**
  - ⋆ Rust (v1.91)

- ⋆ C++ (Standard C++17)
- ⋆ Python 3.12
- ⋆ Windows 10 Pro 22H2 64-bit

- **Środowisko i system budowania:**
  - ⋆ **mach:** Narzędzie wiersza poleceń (CLI) stworzone przez fundacje Mozilla w celu wspomagania rozwoju przeglądarki Firefox, zaimplementowane w języku Python. Pełni ono funkcję generycznego dyspozytora poleceń. W pracy głównym użyciem było uruchamianie budowania biblioteki oraz uruchamiane testów.
  - ⋆ **WSL (Windows Subsystem for Linux):** Wirtualne środowisko wykorzystane pomocniczo do procesu vendorowania bibliotek Rust. Jego użycie było konieczne z powodu na niestabliność mechanizmu pobierania zależności na Windowsie.

- **Narzędzia umożliwiające interoperacyjność:**
  - ⋆ **Bindgen-cli (v0.72.1):** Narzędzie wiersza poleceń służące do automatycznego generowania wiązań (ang. *bindings*) dla języka Rust na podstawie nagłówków C++. Zdecydowano się na statyczne wygenerowanie plików zamiast generowania ich w czasie kompilacji. Jest to zgodne z praktyką stosowaną w projekcie Firefox[19]. Podejście to jest rekomendowane w sytuacjach, gdy natywne API jest stabilne, co pozwala uniknąć niepotrzebnego wydłużania czasu kompilacji projektu.

- **Biblioteki Rust(Crate):**
  - ⋆ **rust-minidump v(0.24):** Kolekcja bibliotek służąca do odczytu i analizy plików zrzutu pamięci, modelowana na podstawie Google Breakpad[20]. W celu przeprowadzenia migracji użyta została biblioteka minidump-processor.
  - ⋆ **sadness-generator (v0.6):** Crate zawierający implementacje kodu wywołującego awarie procesu. Użyty został zgodnie z sugestią zgłaszającego bug 1798688. W praktyce jednak ze względu na specyficzne dla ekosystemu Mozilli mechanizmy obsługi błędów i zarządzania pamięcią, wykorzystanie tej biblioteki zostało ograniczone jedynie do standardowych scenariuszy awarii.
  - ⋆ **windows-sys (v0.52):** Niskopoziomowa biblioteka dostarczająca surowe deklaracje funkcji (FFI), struktur i stałych API systemu Windows, bez dodatkowych warstw abstrakcji. Jej wybór podyktowany był koniecznością bezpośredniego wywoływania funkcji systemowych, co umożliwiło implementację scenariuszy awarii specyficznych dla platformy Windows. **libc (v0.2):** Biblioteka zapewniająca bezpośredni dostęp do standardowych funkcji języka C (np. `malloc`, `free`). Jej użycie było kluczowe przy migracji testów mechanizmu PHC (Probabilistic Heap Checker), gdzie wymagana była surowa alokacja i zwalnianie pamięci z pominięciem bezpiecznych abstrakcji alokatora języka Rust.

- **Narzędzia analityczne i inżynieria wsteczna:**
  - ⋆ **Searchfox:** Narzędzie do semantycznego indeksowania i przeszukiwania kodu źródłowego Mozilli. Wykorzystane do analizy migrowanego kodu oraz identyfikacji miejsc wywołań biblioteki w projekcie. Umożliwiło również poznanie obowiązującego stylu kodowania oraz adaptację istniejących wzorców projektowych.

⋆ **Bugzilla:** System śledzenia błędów pełniący rolę głównego rejestru zadań (ang. *issue tracker*) w organizacji Mozilla. Posłużył do znalezienia fragmentu kodu do migracji oraz analizy wymagań funkcjonalnych.

3.4. MIGRACJA CZĘŚCI ODPOWIEDZIALNEJ ZA ANALIZĘ ZRZUTÓW PAMIĘCI

Proces migracji komponentu odpowiedzialnego za analizę zrzutów pamięci, zaimplementowanego pierwotnie w pliku dumputils.cpp, rozpoczęto od przygotowania środowiska deweloperskiego. Celem było umożliwienie kompilacji kodu Rust i jego integracji z istniejącą, opartą na C++, strukturą biblioteki testcrasher.

3.4.1. PRZYGOTOWANIE ŚRODOWISKA

Integracja nowego kodu w języku Rust z rozbudowanym ekosystemem przeglądarki Firefox wymagała przeprowadzenia kilku operacji konfiguracyjnych w systemie budowania.

Pierwszym krokiem było zdefiniowanie nowego crate - podstawowej jednostki kompilacji i dystrybucji w ekosystemie Rust. W katalogu komponentu utworzono podkatalog rust w ktorym umieszczono trzy pliki:

- **Cargo.toml** - manifest pakietu, definiujący jego metadane (nazwę, wersję, edycję języka "2021") oraz zależności. Skonfigurowano go tak, aby generował bibliotekę statyczną crate-type = ["staticlib"], co jest niezbędne do późniejszego linkowania z kodem C++.
- **lib.rs** - punkt wejścia do biblioteki, początkowo zawierający jedynie szkielet modułu.
- **moz.build** - plik systemu budowania Mozilli umożliwiający kompilację kodu Rust zgodnie z manifestem pakietu.

W tym crate docelowo ma znaleźć się cała nowa implementacja funkcjonalności biblioteki testcrasher.

Następnie, nowo utworzony crate musiał zostać włączony do głównego workspace projektu Firefox. Workspace w kontekście narzędzia Cargo (menedżera pakietów Rust) to zbiór crate'ów, które są zarządzane i kompilowane wspólnie. Dodanie testcrasher do tej struktury formalnie uczyniło go częścią przeglądarki.

Kolejnym krokiem było zsynchronizowanie zależności dla całego projektu. Wykonano to za pomocą polecenia `cargo update -p workspace-hack`. Polecenie to aktualizuje plik Cargo.lock dla całego workspace, zapewniając spójność wersji wszystkich zależności i umożliwiając systemowi budowania poprawne przetwarzanie nowego komponentu.

Ostatnim, etapem było zmodyfikowanie pliku konfiguracyjnego systemu budowania - moz.build w folderze testcrasher oraz dodanie nowego pliku moz.build w folderze zawierającym kod źródłowy Rust. Pliki moz.build, są skryptami w języku Python, jednak ich działanie jest określone poprzez specjalne zasady określone przez Mozille. Ich celem jest określienie funkcjonalności plików w folderze w którym się znajdują[21]. W przypadku biblioteki testcrasher plik moz.build po modyfikacjach wyglądał następująco:

Listing 3.1. Zawartośc pliku moz.build

```
# -*- Mode: python; indent-tabs-mode: nil; tab-width: 40 -*-
# vim: set filetype=python:
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
FINAL_TARGET = "_tests/xpcshell/toolkit/crashreporter/test"
```

```
7
8  XPCSHELL_TESTS_MANIFESTS += ["unit/xpcshell.toml", "unit_ipc/xpcshell.toml"]
9  if CONFIG["MOZ_PHC"]:
10     XPCSHELL_TESTS_MANIFESTS += ["unit/xpcshell-phc.toml", "unit_ipc/xpcshell-phc.toml"]
11
12 TEST_DIRS += [
13     "gtest",
14 ]
15
16 OS_LIBS += [
17     "bcrypt",
18     "synchronization",
19 ]
20
21 DIRS+= ["rust"]
22
23 USE_LIBS += [
24     "rust_testcrasher",
25 ]
26
27 USE_STATIC_MSVCRT = True
28
29 BROWSER_CHROME_MANIFESTS += ["browser/browser.toml"]
30
31 UNIFIED_SOURCES += [
32     "../google-breakpad/src/processor/basic_code_modules.cc",
33     "../google-breakpad/src/processor/convert_old_arm64_context.cc",
34     "../google-breakpad/src/processor/dump_context.cc",
35     "../google-breakpad/src/processor/dump_object.cc",
36     "../google-breakpad/src/processor/logging.cc",
37     "../google-breakpad/src/processor/minidump.cc",
38     "../google-breakpad/src/processor/pathname_stripper.cc",
39     "../google-breakpad/src/processor/proc_maps_linux.cc",
40     "dumputils.cpp",
41     "nsTestCrasher.cpp",
42 ]
43
44 SOURCES += [
45     "ExceptionThrower.cpp",
46 ]
47
48 if CONFIG["OS_TARGET"] == "WINNT" and CONFIG["TARGET_CPU"] == "x86_64":
49     if CONFIG["CC_TYPE"] not in ("gcc", "clang"):
50         SOURCES += [
51             "win64UnwindInfoTests.asm",
52         ]
53
54 if CONFIG["CC_TYPE"] == "clang-cl":
55     SOURCES["ExceptionThrower.cpp"].flags += [
56         "-Xclang",
57         "-fcxx-exceptions",
58     ]
59 else:
60     SOURCES["ExceptionThrower.cpp"].flags += [
61         "-fexceptions",
62     ]
63
64 if CONFIG["MOZ_PHC"]:
65     DEFINES["MOZ_PHC"] = True
```

```
66
67 GeckoSharedLibrary("testcrasher", linkage="dependent")
68
69 DEFINES["SHARED_LIBRARY"] = "%s%s%s" % (
70     CONFIG["DLL_PREFIX"],
71     LIBRARY_NAME,
72     CONFIG["DLL_SUFFIX"],
73 )
74
75 TEST_HARNESS_FILES.xpcshell.toolkit.crashreporter.test.unit += [
76     "CrashTestUtils.sys.mjs"
77 ]
78 TEST_HARNESS_FILES.xpcshell.toolkit.crashreporter.test.unit_ipc += [
79     "CrashTestUtils.sys.mjs"
80 ]
81
82 include("/toolkit/crashreporter/crashreporter.mozbuild")
83
84 NO_PGO = True
85
86 DEFFILE = "testcrasher.def"
```

Modyfikacje dokonane w pliku moz.build umożliwiają współistnienie kodu C++ oraz Rust w ramach finalnej biblioteki testcrasher.dll. Konfiguracja ta instruuje system budowania, aby dołączył rust_testcrasher.lib - bibliotekę statyczną zawierającą skompilowany kod Rust do finalnej biblioteki testcrasher.dll. Istotnym elementem przyjętej strategii było tymczasowe zachowanie pliku dumputils.cpp oraz zależności od Google Breakpad. Takie podejście umożliwiło stopniowe przepisywanie funkcjonalności oraz weryfikację stabilności rozwiązania poprzez testy funkcjonalne po każdej zmianie. Szczegółowa lista zmian wraz z ich celem znajduje się w tabeli 3.1.

Tabela 3.1. Opis zmian w pliku konfiguracyjntm moz.build budującym biblioteke testcrasher - etap 1

Element	Cel zmiany
DIRS	Wskazanie lokalizacji statycznej biblioteki rust_testcrasher
DEFFILE	Definiuje symbole eksportowane z biblioteki statycznej rust_testcrasher
USE_STATIC_MSVCRT	Flaga użyta aby kod C++, jak i Rust używał tej samej, statycznej wersji biblioteki wykonawczej MSVC
USE_LIBS	Wskazanie linkerowi biblioteki statycznej rust_testcrasher aby skompilowany kod z Rust został dodany do biblioteki finalnej testcrasher.dll
OS_LIBS	Dodanie bibliotek systemowych potrzebnych do działania bibliotece rust_testcrasher

3.4.2. OMÓWIENIE MIGRACJI CZĘŚCI ODPOWIEDZIALNEJ ZA ANALIZĘ W JĘZYKU RUST

W niniejszym podrozdziale przedstawiono i przeanalizowano kluczowe zmiany wynikające z migracji kodu z C++ na Rust.

ZASTĄPIENIE ZALEŻNOŚCI I NOWY MODEL OBSŁUGI BŁĘDÓW

- Kod w C++:

```
1 Minidump dump(dump_file);
2 if (!dump.Read()) return false;
3
4 MinidumpException* exception = minidump.GetException();
```

```
if (!exception) {
    return false;
}
```

- Kod w Rust:

```
let addr_str = match fs::read_to_string("crash-addr") {
    Ok(s) => s,
    Err(_) => return false,
};

let addr_str = addr_str.trim();
let addr: u64 = match u64::from_str_radix(&addr_str[2..], 16) {
    Ok(a) => a,
    Err(_) => return false,
};
```

W wersji C++ funkcja dump.Read() zwraca wartość logiczną, a metody GetException() i GetMemoryList() zwracają surowe wskaźniki. Programista jest odpowiedzialny za ręczne sprawdzanie wartości false lub wskaźnika nullptr po każdej operacji. Pominięcie takiego sprawdzenia może prowadzić do niezdefiniowanego zachowania lub awarii programu. Wersja w Ruście wykorzystuje typ Result, który jest typem wyliczeniowym (enum) mogącym przyjąć jeden z dwóch wariantów: Ok(wartość) w przypadku sukcesu lub Err(błąd) w przypadku porażki. Użycie konstrukcji match zmusza programistę do jawnego obsłużenia obu scenariuszy już na etapie kompilacji. Eliminuje to całą klasę błędów polegających na zignorowaniu niepowodzenia operacji. Kompilator Rusta nie pozwoliłby na użycie zmiennej dump lub exception bez uprzedniego "rozpakowania" jej z wariantu Ok.

BEZPIECZEŃSTWO PAMIĘCI

Kod w C++ operuje na surowych wskaźnikach const char* przekazywanych z zewnątrz. Rust, mimo że również musi obsłużyć te wskaźniki w interfejsie FFI (Foreign Function Interface), natychmiast opakowuje je w bezpieczne abstrakcje.

- Kod w C++:

```
extern "C" NS_EXPORT bool DumpHasStream(const char* dump_file, ...) {
  Minidump dump(dump_file);
  // ...
}
```

- Kod w Rust:

```
#[no_mangle]
pub extern "C" fn DumpHasStream(dump_file: *const c_char, ...) -> bool {
    let path_cstr = unsafe {CStr::from_ptr(dump_file)};
    let path_str = match path_cstr.to_str() {
        Ok(s) => s,
        Err(_) => return false,
    };
    // ...
}
```

W C++ const char* nie przechowuje informacji o swojej długości i jest podatny na błędy odczytu poza alokowaną pamięcią. W Ruście surowy wskaźnik *const c_char jest natychmiast konwertowany na bezpieczny typ &CStr. Co istotne, operacja ta jest zamknięta w bloku unsafe, co jasno sygnalizuje miejsce, gdzie programista bierze na siebie odpowiedzialność za poprawność wskaźnika. Zaraz po tym następuje próba konwersji do bezpiecznego, zarządzanego przez Rusta typu &str, która dodatkowo weryfikuje poprawność kodowania UTF-8. Cały pozostały kod funkcji operuje już na bezpiecznych typach, minimalizując powierzchnię ataku.

OPERACJE PLIKOWE I PARSOWANIE

W funkcji DumpCheckMemory widoczna jest znacząca różnica w sposobie obsługi operacji wejścia/wyjścia oraz parsowania danych

- Kod w C++:

```
void* addr;
FILE* fp = fopen("crash-addr", "r");
if (!fp) return false;
if (fscanf(fp, "%p", &addr) != 1) {
  fclose(fp);
  return false;
}
fclose(fp);
```

- Kod w Rust:

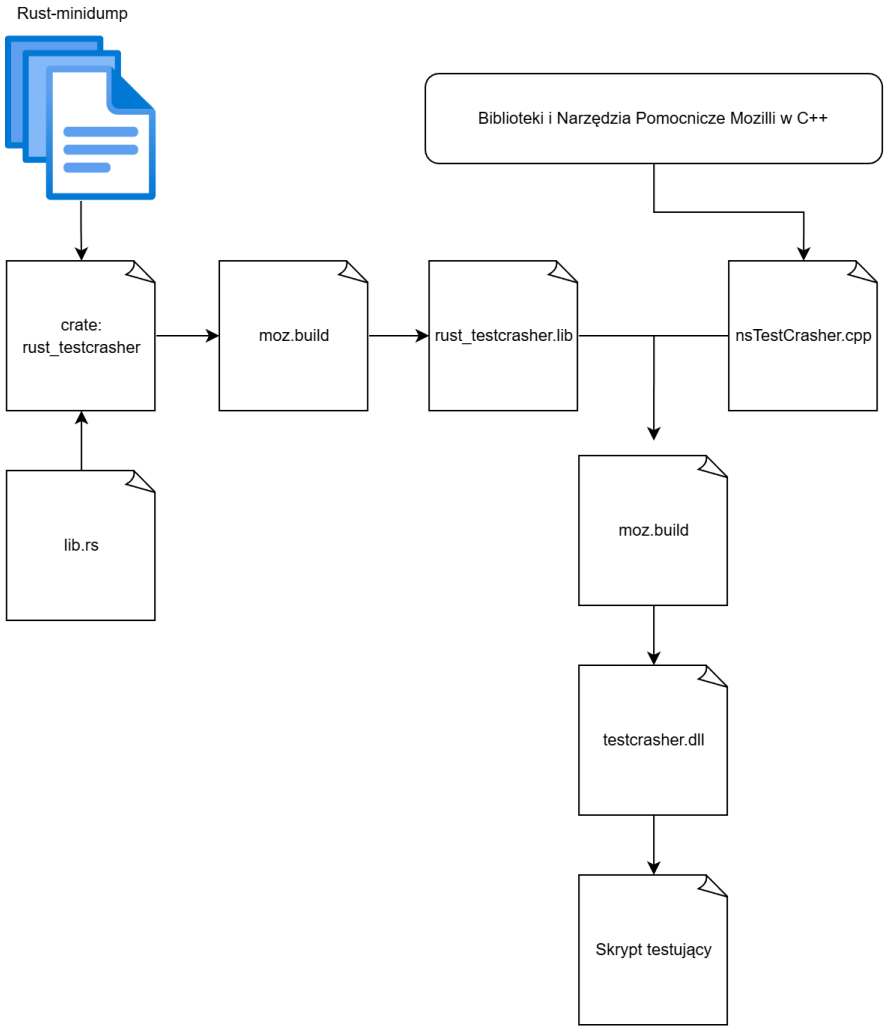
```
let addr_str = match fs::read_to_string("crash-addr") {
    Ok(s) => s,
    Err(_) => return false,
};

let addr_str = addr_str.trim();
let addr: u64 = match u64::from_str_radix(&addr_str[2..], 16) {
    Ok(a) => a,
    Err(_) => return false,
};
```

Podejście z C++ jest niskopoziomowe i podatne na błędy. Programista musi ręcznie zarządzać uchwytem do pliku (FILE*), pamiętając o jego zamknięciu (fclose) w każdej możliwej ścieżce wyjścia z funkcji. Biblioteka standardowa Rusta oferuje znacznie bezpieczniejsze abstrakcje. Funkcja fs::read_to_string w jednej operacji otwiera plik, odczytuje całą jego zawartość do String i automatycznie go zamyka, a wynik zwraca jako Result. Parsowanie jest również jawne i bezpieczne: u64::from_str_radix wymaga podania podstawy systemu liczbowego (tutaj 16) i również zwraca Result, co chroni przed niepoprawnym formatem danych w pliku. Takie podejście jest nie tylko bezpieczniejsze, ale również czytelniejsze.

3.4.3. ARCHITEKTURA PO MIGRACJI MODUŁU ANALITYCZNEGO

W tym kroku, po pomyślnej reimplementacji logiki analizy zrzutów pamięci, możliwe stało się całkowite wyeliminowanie zależności od biblioteki Google Breakpad oraz usunięcie pliku źródłowego dumputils.cpp. Funkcjonalność ta została w pełni przejęta przez kod w języku Rust. Uproszczoną strukturę biblioteki, pozbawioną już komponentów analitycznych napisanych w C++, przedstawiono na schemacie blokowym 3.2. Widać na nim, że plik moz.build nie odwołuje się już

do źródeł Breakpada, a jedynie do biblioteki rust_testcrasher oraz pozostawionego jeszcze pliku nsTestCrasher.cpp.



Rys. 3.2. Architektura biblioteki testcrasher - Etap 2 - opracowanie własne

3.5. MIGRACJA FRAGMENTU ODPOWIEDZIALNEGO ZA WYWOŁYWANIE AWARII PROCESU

Kolejny etap prac obejmował migrację fragmentu kodu którego zadaniem było wywoływanie awarii procesu, logika za to odpowiedzialna orginalnie była zaimplementowana z pliku nsTestCrasher.cpp. Ta część migracji w odróżnieniu od poprzedniej nie zakłada zastąpienia zależności napisanej w języku C++. Podobnie jak w poprzednim etapie migracji trzeba było rozpocząć od przygotowania systemu budowania Mozilli.

3.5.1. PRZYGOTOWANIE ŚRODOWISKA

Prace nad migracją fragmentu wymagały zmian w konfiguracji zarówno pliku systemu budowania moz.build oraz w pliku manifestu pakietu Cargo.toml. Głównym celem było dodanie nowej zależności do projektu Mozilla jakią był crate sadness-generator który zapewniał implementacje niektórych awarii procesu.

DODANIE ZALEŻNOŚCI I PROCES WERYFIKACJI

Procedura dodania biblioteki sadness-generator przebiegała dwuetapowo:

1. **Deklaracja wersji:** Pierwszym krokiem była modyfikacja pliku Cargo.toml w katalogu komponentu. W przypadku bibliotek, które już znajdują się w drzewie projektu, wystarczy wskazać ich wersję.
2. **Vendorowanie (Vendor-ing):** Ponieważ sadness-generator był nową zależnością, nieobecną w repozytorium, konieczne było pobranie jej kodu źródłowego i dołączenie go do projektu. Służy do tego polecenie:

   ```
   ./mach vendor rust
   ```

Podczas realizacji tego kroku napotkano istotne wyzwanie związane ze środowiskiem pracy. Komenda `mach vendor rust` wykazywała niestabilność na systemie Windows, co jest znanym problemem w systemie śledzenia błędów Bugzilla[22]. Aby kontynuować prace, proces vendorowania przeprowadzono w środowisku WSL (Windows Subsystem for Linux), co pozwoliło na poprawne pobranie źródeł.

FLAGA MOZ_PHC

Istotnym wyzwaniem była synchronizacja ustawień kompilacji między systemem budowania Mozilli a Cargo. Mechanizm PHC (Probabilistic Heap Checker) jest opcjonalny i włączany globalną flagą MOZ_PHC. Kod w Rust nie ma jednak bezpośredniego dostępu do zmiennych środowiskowych procesu budowania. Aby rozwiązać ten problem, zastosowano mechanizm Rust features. W pliku moz.build dodano logikę, która sprawdza globalną konfigurację i warunkowo aktywuje odpowiednią flagę dla kompilatora Rust:

Listing 3.2. Definicja feature w Cargo.toml

```
[features]
    moz_phc = []
```

Listing 3.3. Warunkowe włączenie feature w moz.build

```
features = []
if CONFIG["MOZ_PHC"]:
    features += ["moz_phc"]
```

Dzięki temu zabiegowi, w kodzie źródłowym Rust możliwe stało się użycie atrybutu `#[cfg(feature = "moz_phc")]`, aby warunkowo kompilować fragmenty kodu odpowiedzialne za współpracę z PHC, tylko wtedy, gdy mechanizm ten jest aktywny w przeglądarce.

PRZYGOTOWANIE WARSTWY POŚREDNICZĄCEJ (FFI)

Dodatkowo w tym przypadku niemożliwe było przeprowadzenie kompletnej migracji na język Rust. Spowodowane było to faktem że wiele fundamentalnych funkcji, takich jak makro `MOZ_CRASH()` (służące do bezpiecznego przerywania działania programu w krytycznych momentach), jest zaimplementowanych jako makra preprocesora C++ lub funkcje `inline`. Język Rust, komunikując się poprzez interfejs FFI (Foreign Function Interface), może łączyć się jedynie z istniejącymi symbolami, a nie z dyrektywami preprocesora, które znikają po etapie kompilacji C++. W związku z tym, aby zmapować mechanizm `panic!()` w Rust na systemowe wywołanie awarii w Firefoxie, konieczne było zastosowanie wzorca projektowego Shim (adaptera). Stworzono dedykowaną

warstwę kodu łączącego, umożliwia wykorzystanie kodu C++ przez kod w Rust. W tym celu utworzono dwa pliki:

- **wrapper.h** - uproszczony plik nagłówkowy, przygotowany specjalnie dla narzędzia bindgen. Zdefiniowano w nim tylko te typy i funkcje, które są niezbędne dla Rusta, ukrywając resztę złożoności nagłówków C++.

Listing 3.4. wrapper.h

```
/*
bindgen wrapper.h -o rust/src/bindings.rs --enable-cxx-namespaces \
--opaque-type "mozilla::phc::AddrInfo" -- -x c++ -std=c++17
*/

#define MOZ_JEMALLOC_API

namespace mozilla {
namespace phc {

// The types here must match the ones in memory/build/PHC.h
class AddrInfo;

enum PHCState {
  OnlyFree,
  Enabled,
};

} // namespace phc
} // namespace mozilla

extern "C" {
    void Rust_SetPHCState(mozilla::phc::PHCState aState);
    bool Rust_IsPHCAllocation(const void* aPtr, mozilla::phc::AddrInfo* aOutInfo);
    void PureVirtualCall();
    void ThrowException();
    void* Rust_moz_xmalloc(size_t size);
    void Rust_MOZ_CRASH();
}
```

- **wrapper.cpp** - plik implementacyjny, który pełni dwie funkcje:

  1. Udostępnia funkcje pośredniczące (wrappery) do mechanizmów Mozilli, np. `Rust_moz_xmalloc` czy `Rust_SetPHCState`.
  2. Implementuje specyficzne dla C++ scenariusze błędów, niemożliwe do pełnego odwzorowania w Rust (np. funkcja `PureVirtualCall` wywołująca błąd czystej funkcji wirtualnej).

Listing 3.5. wrapper.cpp

```
#include "PHC.h"
#include "mozilla/mozalloc.h"
#include "mozilla/Assertions.h"

extern "C" {

void Rust_SetPHCState(mozilla::phc::PHCState aState) {
  mozilla::phc::SetPHCState(aState);
```

```
}

bool Rust_IsPHCAllocation(const void* aPtr, mozilla::phc::AddrInfo* aOutInfo) {
  return mozilla::phc::IsPHCAllocation(aPtr, aOutInfo);
}

void *Rust_moz_xmalloc(size_t size){
    return moz_xmalloc(size);
}

void Rust_MOZ_CRASH(){
    MOZ_CRASH();
}

/*
 * This pure virtual call example is from MSDN
 */
class A;

void fcn(A*);

class A {
 public:
  virtual void f() = 0;
  A() { fcn(this); }
};

class B : A {
  void f() override {}

 public:
  void use() {}
};

void fcn(A* p) { p->f(); }

void PureVirtualCall() {
  // generates a pure virtual function call
  B b;
  b.use();  // make sure b's actually used
}

}
```

Takie podejście zapewniło czystą separację i pozwoliło uniknąć błędów kompilacji, które pojawiały się przy próbie bezpośredniego parsowania pełnych nagłówków silnika Gecko przez narzędzie bindgen. Rezultatem wykonania komendy `bindgen` był plik bindings.rs, który zawierał interfejs programistyczny (API) zrozumiany dla kompilatora Rust. Należy jednak podkreślić, że plik ten nie zawierał faktycznej implementacji (ciał funkcji), a jedynie ich deklaracje zamknięte w blokach `extern "C"`. Z perspektywy języka Rust są to tzw. symbole zewnętrzne - obietnica, że kod maszynowy tych funkcji zostanie dostarczony później. Fizyczne powiązanie tych deklaracji z ich wykonawczymi odpowiednikami (zaimplementowanymi w wrapper.cpp) następuje dopiero w fazie linkowania. Wówczas linker łączy skompilowany kod Rusta ze skompilowanym obiektem C++, rozwiązując puste symbole i tworząc działającą bibliotekę. Wygenerowany fragment pliku bindings.rs przedstawiono na listingu poniżej:

Listing 3.6. bindings.rs

```
/* automatically generated by rust-bindgen 0.72.1 */

#[allow(non_snake_case, non_camel_case_types, non_upper_case_globals)]
pub mod root {
    #[allow(unused_imports)]
    use self::super::root;
    pub mod mozilla {
        #[allow(unused_imports)]
        use self::super::super::root;
        pub mod phc {
            #[allow(unused_imports)]
            use self::super::super::super::root;
            #[repr(C)]
            #[derive(Debug, Copy, Clone)]
            pub struct AddrInfo {
                _unused: [u8; 0],
            }
            pub const PHCState_OnlyFree: PHCState = 0;
            pub const PHCState_Enabled: PHCState = 1;
            pub type PHCState = ::std::os::raw::c_int;
        }
    }
    unsafe extern "C" {
        pub fn Rust_SetPHCState(aState: root::mozilla::phc::PHCState);
    }
    unsafe extern "C" {
        pub fn Rust_IsPHCAllocation(
            aPtr: *const ::std::os::raw::c_void,
            aOutInfo: *mut root::mozilla::phc::AddrInfo,
        ) -> bool;
    }
    unsafe extern "C" {
        pub fn PureVirtualCall();
    }
    unsafe extern "C" {
        pub fn ThrowException();
    }
    unsafe extern "C" {
        pub fn Rust_moz_xmalloc(size: usize) -> *mut ::std::os::raw::c_void;
    }
    unsafe extern "C" {
        pub fn Rust_MOZ_CRASH();
    }
}
```

3.5.2. OMÓWIENIE MIGRACJI CZĘŚCI ODPOWIEDZIALNEJ ZA WYWOŁANIE AWARII PROCESU W JĘZYKU RUST

W niniejszym podrozdziale przedstawiono i przeanalizowano kluczowe zmiany wynikające z migracji logiki odpowiedzialnej za wywoływanie awarii procesu z pliku nsTestCrasher.cpp na język Rust. Migracja ta wymagała zastosowania innego podejścia niż w przypadku modułu analitycznego, ze względu na konieczność integracji z niskopoziomowymi mechanizmami systemu operacyjnego oraz specyficznymi konstrukcjami języka C++.

STRUKTURA DYSPOZYTORA AWARII

Główna funkcja `Crash()` pełni rolę dyspozytora, który na podstawie przekazanego argumentu wywołuje odpowiedni scenariusz błędu. Porównanie implementacji w obu językach ukazuje fundamentalne różnice w sposobie obsługi wielowariantowej logiki.

- Kod w C++:

Listing 3.7. Dyspozycja awarii w C++ (nsTestCrasher.cpp)

```
extern "C" NS_EXPORT void Crash(int16_t how) {
  switch (how) {
    case CRASH_INVALID_POINTER_DEREF: {
      volatile int* foo = (int*)0x42;
      *foo = 0;
      break;
    }
    case CRASH_PURE_VIRTUAL_CALL: {
      PureVirtualCall();
      break;
    }
    case CRASH_OOM: {
      [[maybe_unused]] void* r0 = moz_xmalloc((size_t)-1);
      [[maybe_unused]] void* r1 = moz_xmalloc((size_t)-1);
      [[maybe_unused]] void* r2 = moz_xmalloc((size_t)-1);
      break;
    }
    // inne przypadki
  }
}
```

- Kod w Rust:

Listing 3.8. Dyspozycja awarii w Rust (ns_test_crasher.rs)

```
#[no_mangle]
pub unsafe extern "C" fn Crash(how: i16) {
    match how {
        CRASH_INVALID_POINTER_DEREF => {
            SadnessFlavor::Segfault.make_sad();
        },
        CRASH_PURE_VIRTUAL_CALL => {
            PureVirtualCall();
        },
        CRASH_OOM => {
            Rust_moz_xmalloc(usize::MAX);
            Rust_moz_xmalloc(usize::MAX);
            Rust_moz_xmalloc(usize::MAX)
        },
        // inne przypadki
    }
}
```

W wersji C++ wykorzystano tradycyjną instrukcję `switch-case`, która wymaga jawnego użycia słowa kluczowego `break` po każdym przypadku. W języku Rust konstrukcja `match` jest wyrażeniem, które wymusza pełne pokrycie wszystkich możliwych wariantów. Kompilator zgłasza błąd, jeśli programista nie obsłuży wszystkich przypadków. Dodatkowo, każde ramię konstrukcji match jest izolowane i nie występuje problem przepływu, jak w przypadku C++ przy pominięciu `break`.

Istotną różnicą jest również wykorzystanie biblioteki `sadness-generator` do implementacji standardowych scenariuszy awarii. Zamiast ręcznie konstruować wskaźniki jak w C++ (`volatile int* foo = (int*)0x42`), Rust wykorzystuje abstrakcję `SadnessFlavor::Segfault.make_sad()`, która jest bezpieczniejsza i bardziej czytelna.

ZARZĄDZANIE PAMIĘCIĄ W FUNKCJI SAVEAPPMEMORY

Funkcja `SaveAppMemory()` demonstruje fundamentalną różnicę w podejściu do zarządzania pamięcią między C++ a Rust. Jej zadaniem jest alokacja bufora, wypełnienie go danymi testowymi i zapisanie adresu do pliku.

- Kod w C++:

Listing 3.9. Zapisywanie pamięci aplikacji w C++ (nsTestCrasher.cpp)

```cpp
char testData[32];

extern "C" NS_EXPORT uint64_t SaveAppMemory() {
  for (size_t i = 0; i < sizeof(testData); i++)
    testData[i] = i;

  FILE* fp = fopen("crash-addr", "w");
  if (!fp) return 0;
  fprintf(fp, "%p\n", (void*)testData);
  fclose(fp);

  return (int64_t)testData;
}
```

- Kod w Rust:

Listing 3.10. Zapisywanie pamięci aplikacji w Rust (ns_test_crasher.rs)

```rust
#[no_mangle]
pub extern "C" fn SaveAppMemory() -> u64 {
    let mut test_data = Box::new([0u8; 32]);
    for i in 0..32 {
        test_data[i] = i as u8;
    }
    let leaked_ref: &'static mut [u8; 32] = Box::leak(test_data);
    let addr = leaked_ref.as_ptr() as u64;

    if let Ok(mut file) = fs::File::create("crash-addr") {
        let _ = writeln!(file, "0x{:x}", addr);
    }
    addr
}
```

W C++ bufor `testData` jest zmienną globalną o statycznym czasie życia, co oznacza że pamięć jest alokowana na etapie kompilacji i istnieje przez cały czas działania programu. Podejście to jest proste, ale wprowadza stan mutowalny, który może być źródłem błędów.

Wersja w Rust wykorzystuje mechanizm `Box::leak()`, który celowo "wycieka"pamięć, tworząc referencję o statycznym czasie życia. Pamięć musi przetrwać do momentu awarii procesu, aby moduł Crashreporter mógł ją odczytać. Użycie `Box::leak()` jest jawną deklaracją, w przeciwieństwie do globalnego stanu w C++. w Rust nie ma zmiennych globalnych o statycznym czasie

życia, które można mutować bezpiecznie (poza `static mut`, ale jest to niezalecane i niebezpieczne). Dodatkowo, zamiast C++ funkcji `fopen/fprintf/fclose` z ręcznym sprawdzaniem błędów, w języku Rust zastosowano `fs::File::create()` z konstrukcją `if let Ok(...)`, która obsługuje potencjalne błędy.

KOMPILACJA WARUNKOWA I MECHANIZM PHC

Migracja kodu związanego z mechanizmem PHC wymagała zastosowania odpowiedników dyrektyw preprocesora C++ w języku Rust.

- Kod w C++:

Listing 3.11. Alokacja PHC w C++ (nsTestCrasher.cpp)

```
#ifdef MOZ_PHC
uint8_t* GetPHCAllocation(size_t aSize) {
  for (int i = 0; i < 2000000; i++) {
    uint8_t* p = (uint8_t*)malloc(aSize);
    if (mozilla::phc::IsPHCAllocation(p, nullptr)) {
      return p;
    }
    free(p);
  }
  MOZ_CRASH("failed to get a PHC allocation");
}
#endif
```

- Kod w Rust:

Listing 3.12. Alokacja PHC w Rust (ns_test_crasher.rs)

```
#[cfg(feature = "moz_phc")]
#[no_mangle]
pub extern "C" fn GetPHCAllocation(size: usize) -> *mut u8 {
    for _ in 0..2000000 {
        unsafe {
            let p = malloc(size) as *mut u8;
            if !p.is_null() &&
               Rust_IsPHCAllocation(p as *mut c_void,
                                    std::ptr::null_mut()) {
                return p;
            }
            free(p as *mut c_void);
        }
    }
    panic!("failed to get a PHC allocation");
}
```

Kod Rust korzysta z funkcji biblioteki `libc` (`malloc` i `free`) zamiast bezpiecznych alokatorów Rust. Jest to konieczne, ponieważ mechanizm PHC wymaga bezpośredniego dostępu do alokacji pamięci w sposób, który może być monitorowany przez moduł PHC. Użycie standardowych abstrakcji Rust (np. `Vec` lub `Box`) uniemożliwiło poprawne działanie testów i wymagało dodatkowych zmian w całym projekcie, które nie wchodzą w zakres niniejszej pracy.

INTEGRACJA Z WINDOWS API

Obsługa niskopoziomowych mechanizmów systemu Windows, takich jak rejestracja niestandardowego filtru wyjątków, wymagała precyzyjnego odwzorowania API systemu operacyjnego.

- Kod w C++:

Listing 3.13. Obsługa wyjątków w C++ (nsTestCrasher.cpp)

```cpp
#ifdef XP_WIN
static LONG WINAPI HandleException(EXCEPTION_POINTERS* exinfo) {
  TerminateProcess(GetCurrentProcess(), 0);
  return 0;
}

extern "C" NS_EXPORT void TryOverrideExceptionHandler() {
  SetUnhandledExceptionFilter(HandleException);
}
#endif
```

- Kod w Rust:

Listing 3.14. Obsługa wyjątków w Rust (ns_test_crasher.rs)

```rust
#[cfg(target_os = "windows")]
unsafe extern "system" fn handle_exception(
    _exinfo: *const EXCEPTION_POINTERS
) -> i32 {
    TerminateProcess(GetCurrentProcess(), 0);
    0
}

#[cfg(target_os = "windows")]
#[no_mangle]
pub extern "C" fn TryOverrideExceptionHandler() {
    unsafe {
        SetUnhandledExceptionFilter(Some(handle_exception));
    }
}
```

W C++ integracja z Windows API jest bezpośrednia i wystarczy dołączyć odpowiedni nagłówek (`<windows.h>`), zatem wywołać funkcje systemowe. Konwencja wywołania (`WINAPI`) i typy (np. `LONG`, `EXCEPTION_POINTERS*`) są natywnie obsługiwane.

W Rust konieczne było użycie biblioteki `windows-sys`, która dostarcza niskopoziomowe wiązania do Windows API. Funkcja obsługująca wyjątki musi być oznaczona konwencją wywołania `extern "system"`, która na platformie Windows odpowiada konwencji `stdcall`. Atrybut `#[cfg(target_os = "windows")]` zapewnia, że kod jest kompilowany wyłącznie na platformie docelowej.

OBSŁUGA TESTÓW CFI NA ARCHITEKTURZE X86_64

Następnym wyzwaniem była migracja logiki odpowiedzialnej za testy CFI, które wymagają bezpośredniej interakcji z kodem asemblerowym.

Listing 3.15. Rozwiązywanie adresów funkcji CFI w Rust

```rust
#[cfg(all(windows, target_arch = "x86_64"))]
type CfiAsmFunc = unsafe extern "C" fn(u64, *mut c_void) -> u64;

#[cfg(all(windows, target_arch = "x86_64"))]
unsafe fn resolve_cfi_func_addr(fnid: i16) -> u64 {
    let target_func: CfiAsmFunc = match fnid {
        CRASH_X64CFI_NO_MANS_LAND => x64CrashCFITest_NO_MANS_LAND,
```

```
8           CRASH_X64CFI_LAUNCHER => x64CrashCFITest_Launcher,
9           CRASH_X64CFI_UNKNOWN_OPCODE => x64CrashCFITest_UnknownOpcode,
10          // inne przypadki
11          _ => return 0,
12      };
13      // Adresy wskazuja na tablice skokow, nie na ciala funkcji.
14      // Pobranie poprawnego wskaznika przez wywolanie z returnpfn=1
15      target_func(1, std::ptr::null_mut())
16  }
```

Funkcje testowe CFI są zaimplementowane w języku asemblera (plik `win64UnwindInfoTests.asm`) i nie podlegały migracji. Zadaniem kodu Rust było jedynie zapewnienie interfejsu do ich wywoływania. W tym celu zdefiniowano blok `extern "C"` z deklaracjami funkcji zewnętrznych oraz funkcję pomocniczą `resolve_cfi_func_addr()`, która mapuje identyfikatory na wskaźniki do funkcji asemblerowych.

Warto zwrócić uwagę na wyjaśniający komentarz w kodzie: funkcje asemblerowe zwracają adresy wskazujące na tablicę skoków, nie na właściwe ciała funkcji. Aby uzyskać poprawny wskaźnik, należy wywołać funkcję z argumentem `returnpfn=1`. Ten mechanizm został zachowany bez zmian z oryginalnej implementacji C++.

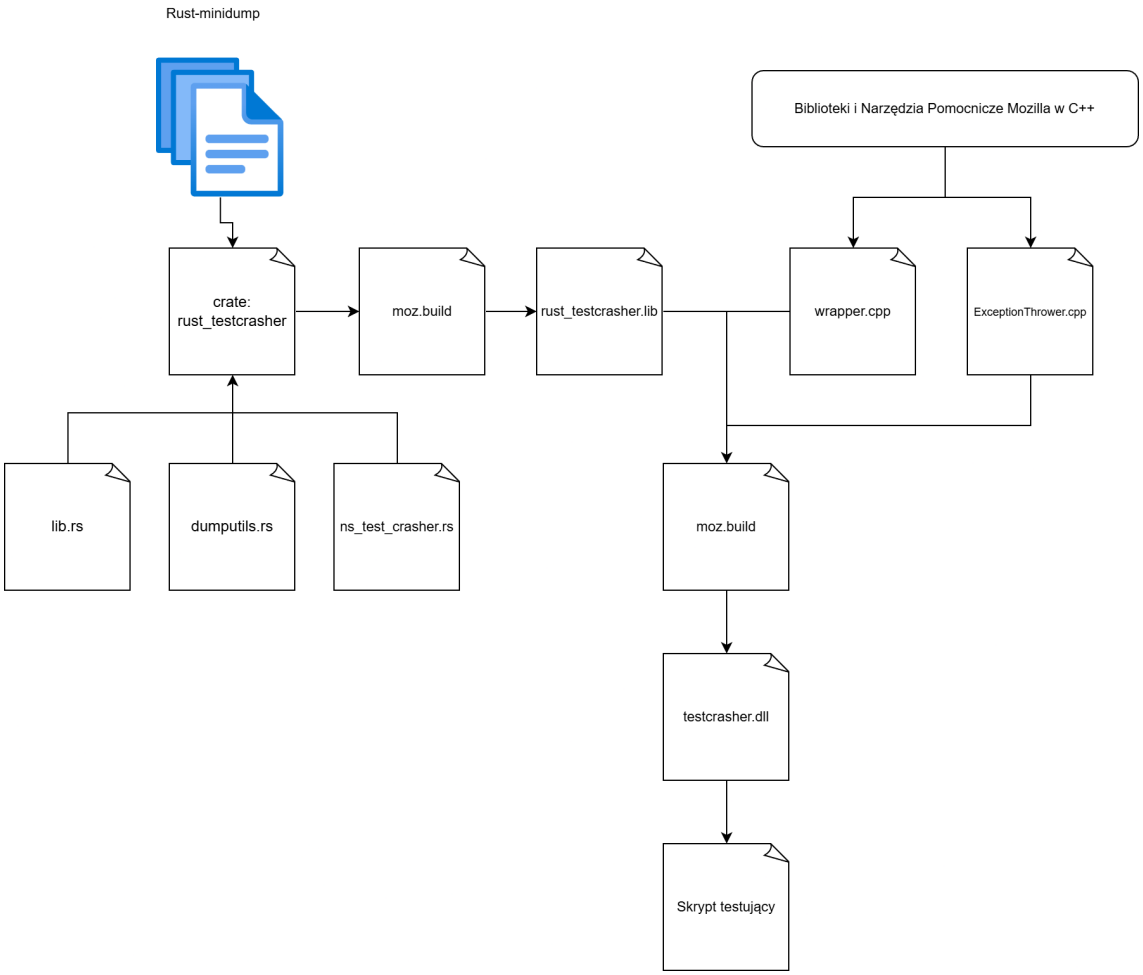
3.5.3. ARCHITEKTURA PO MIGRACJI

Po zakończeniu procesu migracji obu modułów(analitycznego (dumputils.cpp) oraz odpowiedzialnego za wywoływanie awarii (nsTestCrasher.cpp)), architektura biblioteki testcrasher uległa znacznej zmianie. Całość logiki funkcjonalnej została przeniesiona do kodu napisanego w języku Rust, eliminując zależność od biblioteki Google Breakpad oraz znaczną część oryginalnego kodu C++. Mimo przeprowadzenia migracji, pewne elementy pozostały zaimplementowane w języku oryginalnym:

- **wrapper.cpp** - warstwa pośrednicząca zapewniająca dostęp do mechanizmów niedostępnych bezpośrednio w języku Rust (makro `MOZ_CRASH()`, funkcje PHC, alokator `moz_xmalloc`).

- **ExceptionThrower.cpp** - kod odpowiedzialny za generowanie wyjątków C++, które nie mogą być w pełni odwzorowane w Rust.

- **win64UnwindInfoTests.asm** - kod asemblerowy testów CFI.



Rys. 3.3. Architektura biblioteki testcrasher - Etap 3 - opracowanie własne

Końcowa architektura charakteryzuje się następująco:

1. **Warstwa API (Rust):** Funkcje eksportowane (`Crash()`, `SaveAppMemory()`, `DumpHasStream()` itd.) są zaimplementowane w Rust i stanowią publiczny interfejs biblioteki.
2. **Warstwa pośrednicząca (C++):** Plik `wrapper.cpp` dostarcza funkcje pomocnicze (`Rust_moz_xmalloc()`, `Rust_MOZ_CRASH()`) wywoływane z kodu Rust poprzez FFI.
3. **Zależności zewnętrzne:** Biblioteka `rust-minidump` zastąpiła Google Breakpad do analizy plików zrzutu pamięci, a `sadness-generator` dostarcza implementacje standardowych scenariuszy awarii.

PROCES BUDOWANIA

Proces budowania biblioteki został rozszerzony o etap kompilacji kodu Rust:

1. **Kompilacja Rust:** System budowania Mozilla (`mach`) wywołuje Cargo, który kompiluje kod źródłowy Rust do biblioteki statycznej `rust_testcrasher.lib`.
2. **Kompilacja C++:** Pliki `wrapper.cpp`, `ExceptionThrower.cpp` oraz opcjonalnie `win64UnwindInfoTests.as` są kompilowane do obiektów.
3. **Linkowanie:** Wszystkie komponenty są łączone w finalną bibliotekę dynamiczną `testcrasher.dll`. Symbole zdefiniowane w pliku `testcrasher.def` są eksportowane jako API.

4. TESTOWANIE I WNIOSKI KOŃCOWE

Testy zostały przeprowadzone w następującym środowisku deweloperskim, przedstawionym poniżej:

- System operacyjny: Windows 10 Pro 22H2 64-bit
- Procesor: AMD Ryzen 5 7600X (6 × 4.70 GHz)
- Pamięć RAM: 32 GB
- rustc: 1.91.0
- cargo: 1.91.0
- clang: 19.1.5
- Firefox: 144.0.1 (release)

W ramach testów funkcjonalnych oceniano poprawność działania biblioteki testcrasher po migracji z języka C++ do Rust. Uruchomiono dwa zestawy testów:

- **Testy jednostkowe** `./mach test toolkit/crashreporter/test/unit/`
- **Testy IPC** `./mach test toolkit/crashreporter/test/unit_ipc/`

Testom poddano dwa główne obszary funkcjonalne biblioteki: analiza zrzutu pamięci: **DumpHasStream()**, **DumpHasInstructionPointerMemory()**, **DumpCheckMemory()** oraz wywoływanie awarii procesu: **Crash()**, **EnablePHC()**, **GetWin64CFITestFnAddrOffset()**, **TryOverrideExceptionHandler()**, **SaveAppMemory()**. Dodatkowo weryfikowano brak regresji, rozumiany jako zachowanie niezmienionego ABI biblioteki, identyczne zachowanie funkcji eksportowanych oraz przejście wszystkich testów XPCShell bez modyfikacji po stronie testów. W ramach analizy porównano również czas kompilacji oraz rozmiar wygenerowanej biblioteki przed i po migracji.
Do obliczenia czasu kompilacji oraz rozmiaru pliku biblioteki testcrasher zostały użyte następujące metody pomiarowe: czas kompilacji, raportowany przez `mach`, oraz rozmiar pliku, raportowany przez PowerShell: `(Get-Item "testcrasher.dll").Length`.

4.1. WERYFIKACJA FUNKCJONALNA

Wyniki testów przed i po migracji były identyczne, co potwierdza, że implementacja w języku Rust zachowała pełną kompatybilność funkcjonalną z wersją C++.

- Testy jednostkowe:

```
Ran 48 checks (48 tests)
Expected results: 43
Skipped: 4 tests
Unexpected results: 1 (1 fail)
```

- Testy IPC:

```
Ran 10 checks (10 tests)
Expected results: 10
Unexpected results: 0
OK
```

Takie same wyniki uzyskano po migracji biblioteki na Rust. Oznacza to, że wszystkie ścieżki funkcjonalne zostały zachowane oraz ABI biblioteki pozostało zgodne.

4.2. ANALIZA CZASU KOMPILACJI I ROZMIARU PLIKU

4.2.1. CZAS KOMPILACJI

Poniżej przedstawiono wyniki testów porównawczych dotyczących czasu kompilacji biblioteki testcrasher przed i po migracji:

Tabela 4.1. Czas kompilacji biblioteki `testcrasher` przed i po migracji

Wersja	Czas	Zmiana
Przed migracją	0:00.90	–
Po migracji	0:01.23	+36%

Wzrost czasu wynika bezpośrednio z charakterystyki kompilatora Rust: analiza typów jest znacząco bardziej kosztowna obliczeniowo, Rust stosuje intensywną optymalizację kodu, generowanie jest często wolniejsze niż w przypadku C++.[23] Choć czas kompilacji wzrósł o 36%, jest to niewielka różnica w skali całej przeglądarki Firefox. W praktyce zdecydowana różnica czasowa staje się pomijalna przy ciągłym rozwoju.

4.2.2. ROZMIAR PLIKU BIBLIOTEKI

Poniżej przedstawiono wyniki testów porównawczych dotyczących rozmiaru pliku biblioteki testcrasher przed i po migracji:

Tabela 4.2. Rozmiar biblioteki `testcrasher` przed i po migracji

Wersja	Rozmiar	Zmiana
Przed migracją	167 936 B	–
Po migracji	455 680 B	+271%

Znaczne zwiększenie rozmiaru biblioteki wynika z charakterystyki kompilatora Rust, który statycznie dołącza część niezbędnych elementów bibliotecznych, generuje bardziej rozbudowane konstrukcje wewnętrzne, wynikające z modelu bezpieczeństwa pamięci oraz typów opisujących błędy, nie korzysta z udostępnianych przez system współdzielonych bibliotek, co skutkuje większą ilością kodu dołączanego do finalnego pliku w porównaniu do kompilacji C++.[24][25] W praktyce zwiększenie rozmiaru z 167 KB do 455 KB nie ma wpływu na działanie systemu.

4.3. NAPOTKANE PROBLEMY I OGRANICZENIA

W trakcie migracji zidentyfikowano następujące problemy i ograniczenia:

- **Różnice w obsłudze błędów.** W C++ błędy sygnalizowane są przez nullptr i wartości logiczne, podczas gdy w Rust wymuszona jest obsługa Result<T, E> oraz Option<T>.
- **Różnice między Breakpad a rust-minidump.** API rust-minidump nie jest bezpośrednim odpowiednikiem Google Breakpad, co wymagało zmiany sposobu parsowania zrzutów pamięci.
- **Długi czas kompilacji.** Pełna kompilacja projektu na podanym systemie trwa ok. 30 minut. Możliwe jest kompilowanie wyłącznie modułów, jednak nie zawsze jest to wystarczające, szczególnie w przypadku zmian ABI lub plików moz.build.

- **Problemy z narzędziami vendoringu na platformie Windows.** Proces integracji zewnętrznej biblioteki sadness-generator (opisany w rozdziale 3.5.1) wymagał użycia polecenia `mach vendor rust`. Podczas uruchamiania go w natywnym środowisku Windows napotkano błąd polegający na nieprawidłowej normalizacji separatorów ścieżek (zamiana `/` na `\`) oraz zmianie formatowania znaków końca linii (CRLF zamiast LF). Skutkowało to inwalidacją sum kontrolnych w plikach metadanych (`checksum.json`) dla setek niezmienionych plików, co uniemożliwiało poprawne zatwierdzenie zmian w systemie kontroli wersji. Problem rozwiązano poprzez wykonanie operacji w środowisku WSL (Windows Subsystem for Linux), co wymusiło zachowanie formatowania zgodnego ze standardami projektu.

4.4. WNIOSKI KOŃCOWE

Przeprowadzona migracja fragmentu biblioteki testcrasher z języka C++ do Rust pozwoliła na praktyczną weryfikację przedstawionych założeń pracy. Zgodnie z celem pracy, polegającym na ocenie możliwości zastąpienia kodu typu legacy nowoczesnym i bezpieczniejszym kodem Rust, uzyskane wyniki potwierdzają, że taka migracja jest jak najbardziej możliwa oraz uzasadniona. Podsumowując, migracja analizowanego modułu potwierdza, że Rust jest realną i skuteczną alternatywą dla C++, może być wdrażany stopniowo w dużych projektach, zapewnia korzyści w zakresie bezpieczeństwa pamięci i jakości kodu oraz nie zakłóca stabilności ani kompatybilności całego systemu. Otrzymane wyniki potwierdzają zasadność stosowania Rust w projektach o wysokim znaczeniu dla bezpieczeństwa, takich jak Mozilla Firefox, oraz wskazują kierunek dla dalszej modernizacji istniejących komponentów.

5. PODSUMOWANIE

Niniejsza praca miała na celu praktyczną weryfikację możliwości migracji fragmentu istniejącego kodu w języku C++ do języka Rust w kontekście dużego, rozwijanego od wielu lat projektu, jakim jest przeglądarka Mozilla Firefox. Przedmiotem badań została wybrana biblioteka testcrasher, wykorzystywana w procesie testowania mechanizmów raportowania awarii. Komponent ten jest istotnym elementem zestawu narzędzi deweloperskich wykorzystywanych w projekcie, a jednocześnie na tyle izolowanym, aby jego migracja mogła zostać przeprowadzona w sposób bezpieczny i pozwalała na ocenę praktycznych wyników.

Przeprowadzona analiza istniejących rozwiązań wskazała, że Rust od kilku lat zyskuje na popularności jako język zapewniający wysoki poziom bezpieczeństwa pamięci oraz poprawę stabilności dużych systemów. Projekty takie jak Stylo, Servo czy inicjatywy Mozilli w zakresie Oxidation potwierdzają, że Rust może latwo zastępować fragmenty kodu C++ w złozonych systemach. Jednocześnie narzędzia takie jak C2Rust, Bindgen czy Corrode wspierają proces stopniowego wdrażania języka Rust oraz utrzymania kompatybilności z istniejącymi komponentami.

Proces migracji biblioteki testcrasher obejmował kluczowe elementy jej działania, a głównym wyzwaniem była konieczność zachowania pełnej zgodności funkcjonalnej oraz zachowanie stabilnosci binarnej(ABI). Zastosowana iteracyjna, stopniowa strategia umożliwiła płynne zastępowanie funkcji C++ kodem Rust.

Język programowania Rust eliminuje wiele problemów typowych dla C++, w szczególności związanych z zarządzaniem pamięcią. To wynika z modelu własności oraz systemu typów wymuszającego jawne traktowanie błędów. Testy funkcjonalne wykazały pełną zgodność działania implementacji po migracji z wersją oryginalną, a różnice w rozmiarze biblioteki i czasie kompilacji nie wpływają na funkcjonalność systemu.

Zrealizowana praca dowodzi, że stopniowa migracja modułów C++ do Rust w dużym systemie jest możliwa i przynosi wymierne korzyści. Rust zwiększa bezpieczeństwo pamięci oraz minimalizuje ryzyko błędów trudnych do wykrycia w językach niższego poziomu. Wyniki te potwierdzają wybrany w ostatnich latach przez Mozilla oraz inne organizacje kierunek. W przyszłości biblioteka testcrasher zaimplementowana w Rust mogłaby zostać rozszerzona o kolejne warstwy funkcjonalne. Dalsza integracja Rust w większej liczbie modułów Firefox mogłaby zwiększyć stabilność i jakość całego oprogramowania. Wyniki przedstawione w pracy stanowią podstawę do kontynuacji badań nad wdrażaniem Rust w wieloletnich i dużych projektach programistycznych.

WYKAZ LITERATURY

[1] Stack Overflow. *The 2025 Developer Survey*. https://survey.stackoverflow.co/2025/technology. 2025.

[2] C. K. Alex Rebert. *Secure by Design: Google's Perspective on Memory Safety*. https://storage.googleapis.com/gweb-research2023-media/pubtools/7665.pdf. 2024.

[3] National Security Agency. *Software Memory Safety*. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/1/CSI_SOFTWARE_MEMORY_SAFETY.PDF. 2022.

[4] Galois, Inc. and Immunant, Inc. *C2Rust Manual*. https://c2rust.com/manual. 2023.

[5] The Rust Programming Language. *The bindgen User Guide*. https://rust-lang.github.io/rust-bindgen. 2025.

[6] J. Sharp. *Corrode: C to Rust Translator*. https://github.com/jameysharp/corrode. 2021.

[7] rust-lang. *Unsafe Rust - The Rust Programming Language - Rust Documentation*. https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html. 2025.

[8] Mozilla Foundation. *Quantum/Stylo Project*. https://wiki.mozilla.org/Quantum/Stylo. 2017.

[9] Servo Project. *Servo*. https://github.com/servo/servo. 2025.

[10] Deno Team. *Deno Runtime Documentation*. https://docs.deno.com/runtime/. 2024.

[11] Linux Kernel Developers. *Rust in the Linux Kernel – Quick Start*. https://docs.kernel.org/rust/quick-start.html. 2024.

[12] 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.

[13] R. Ayeshani. *How web browsers use process & Threads ? (Firefox VS Chrome)*. https://randiayeshani.medium.com/how-web-browsers-use-process-threads-305b5a2164d4. 2020.

[14] Wikipedia contributors. *Quantum/Stylo*. https://wiki.mozilla.org/Quantum/Stylo. 2018.

[15] Mozilla Foundation. *Oxidation Initiative*. https://wiki.mozilla.org/Oxidation. 2020.

[16] CXX Project. *CXX: Safe FFI between Rust and C++*. https://cxx.rs/. 2024.

[17] Mozilla Build Team. *Rust in Firefox Build System*. https://firefox-source-docs.mozilla.org/build/buildsystem/rust.html. 2024.

[18] G. Svelto. *Replace breakpad with rust-minidump in the testcrasher library*. https://bugzilla.mozilla.org/show_bug.cgi?id=1798688. 2022.

[19] M. Goregaokar. *Integrating Rust and C++ in Firefox*. https://manishearth.github.io/blog/2021/02/22/integrating-rust-and-c-plus-plus-in-firefox. 2021.

[20] rust-minidump. *rust-minidump*. https://github.com/rust-minidump/rust-minidump. 2025.

[21] Mozilla Foundation. *moz.build Files*. https://firefox-source-docs.mozilla.org/build/buildsystem/mozbuild-files.html. 2025.

[22] B. Tsoi. *[meta] './mach vendor rust' on Windows*. https://bugzilla.mozilla.org/show_bug.cgi?id=1938341. 2024.

[23] The Rust Programming Language. *Overview of the compiler*. https://rustc-dev-guide.rust-lang.org/overview.html. 2025.

[24] The Rust Programming Language. *Linkage*. https://doc.rust-lang.org/reference/linkage.html. 2025.

[25] pnkfx. *Linking Rust Crates*. https://blog.pnkfx.org/blog/2022/05/12/linking-rust-crates. 2022.

WYKAZ RYSUNKÓW

WYKAZ TABEL