JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 19 (2021) 132–138
Received: 19 April 2021
Accepted: 04 May 2021

Compilation of iOS frameworks from Linux operating system using open-source tools

Kompilacja bibliotek iOS w systemie Linux z wykorzystaniem narzędzi open-source

Łukasz Rutkowski*, Piotr Kopniak

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
This paper analyzes possibility of using open-source tools to compile iOS frameworks in Linux operating system. The purpose of this analysis was to determine how compilation in Linux could be performed and identify possible limitations when using LLVM compiler. The analysis has been performed on own frameworks written using Objective-C and Swift languages containing graphic and text files in different formats and sizes. Results of the analysis show that compilation of iOS frameworks under Linux operating system is possible unless the compiler frameworks use interface components written in xib format for which there are no compilation tools available on Linux operating system.

Keywords: iOS frameworks; LLVM compiler; cross-compilation; open-source tools

Streszczenie
W artykule opisano analizę możliwości wykorzystania narzędzi open-source do kompilacji bibliotek iOS w systemie operacyjnym Linux. Celem analizy jest sprawdzenie możliwości przeprowadzenia kompilacji w systemie Linux oraz wykrycie potencjalnych ograniczeń przy wykorzystaniu kompilatora LLVM. Badania przeprowadzono na autorskich bibliotekach napisanych w językach Objective-C oraz Swift, które zawierały pliki graficzne o różnych formatach i rozmiarach, jak również pliki tekstowe. Uzyskano wyniki które wskazują, że kompilacja bibliotek iOS w systemie Linux jest możliwa pod warunkiem, że kompilowane biblioteki nie wykorzystują komponentów opisanych w formacie xib, do kompilacji których na systemie Linux nie istnieje odpowiednik narzędzia kompilacyjnego z systemu macOS.

Słowa kluczowe: biblioteki iOS; kompilator LLVM; kompilacja skośna; narzędzia open-source

*Corresponding author
Email address: lukasz.rutkowski1@pollub.edu.pl (Ł. Rutkowski)

1. Wstęp

Zależności wykorzystywane w aplikacjach iOS, czyli tak zwane biblioteki to tworzony przez programistów zbiór klas, funkcji lub innych konstrukcji programistycznych [1]. Wykorzystywane są one przez twórców aplikacji iOS do współdzielenia rozwiązań pomiędzy różnego rodzaju aplikacjami bądź wydzielana pewnych części kodu do niezależnych zbiorów. Oficjalnie Apple, czyli firma tworząca system iOS udostępnia narzędzia pozwalające na kompilację takich bibliotek wyłącznie na komputerach z systemem operacyjnym macOS [2]. Jednym z takich narzędzi jest środowisko programistyczne Xcode oraz wchodzące w jego skład narzędzie wiersza poleceń xcodebuild [3-4].

Niniejszy artykuł ma na celu zbadanie, czy istnieje możliwość przeprowadzenia procesu kompilacji bibliotek iOS na komputerach z systemem operacyjnym Linux. W pierwszej kolejności przygotowano biblioteki, które zostaną poddane kompilacji. Opisano sposób ich budowania w systemie macOS, a następnie przedstawiono w jaki sposób proces ten może zostać przeprowadzony również na komputerach z systemem operacyjnym Linux. Na koniec porównano skompilowane biblioteki pod kątem czasu ich kompilacji, rozmiaru plików wynikowych oraz wydajności podczas działania w aplikacji końcowej.

2. Przegląd stanu badań

Temat kompilacji bibliotek iOS na systemie Linux nie jest czymś co zostało obszernie omówione przez innych autorów. Najbliżej zbliżonym tematycznie rozwiązaniem jakie udało się znaleźć jest mechanizm kompilacji wykorzystywany przez silnik tworzenia gier Godot [5-6]. Pozwala on na kompilację gier iOS z poziomu systemu operacyjnego Linux. Opisano tutaj instrukcję krok po kroku jakie operacje należy wykonać oraz jakie narzędzia wykorzystać, aby zbudować grę na systemie Linux, którą będzie można następnie uruchamiać na urządzeniach z systemem iOS. Proces ten przeprowadzony został za pośrednictwem zestawu narzędzi cctools-port dostępnego na systemie Linux stanowiących odpowiednik narzędzi udostępnianych przez firmę Apple na systemie macOS [7].

Kolejnym przykładem rozwiązania innych autorów jest proces dołączania do aplikacji iOS dynamicznych bibliotek analizowany w projekcie [8]. Projekt ten dotyczył zagadnienia dołączania do skompilowanych aplikacji bibliotek monitorujących ich zachowanie w trakcie ich działania. Opracowano tutaj sposób na odtworzenie procesu dołączania bibliotek do aplikacji iOS tak, aby działał także na systemie Linux. Dzięki temu pozbyto się wymogu posiadania urządzeń z systemem macOS pozwalając pracownikom naukowym zajmującym się

tematami bezpieczeństwa do wykonywania swojej pracy z wykorzystaniem systemów Linux.

3. Biblioteki poddane analizie

W celu analizy procesu kompilacji bibliotek przeznaczonych dla systemu iOS wykonywanego na systemie macOS, a następnie dostosowania go do działania na systemie Linux przygotowano zestaw bibliotek iOS. Biblioteki te dobrane zostały w taki sposób, aby móc zaobserwować jaki wpływ na działanie całego procesu kompilacji mają poszczególne funkcjonalności bądź zależności, z których mogą korzystać twórcy bibliotek.

Większość z analizowanych bibliotek to biblioteki autorskie przygotowane w taki sposób, aby analizowały pojedynczy aspekt wpływający na proces kompilacji. Napisane zostały one w języku programowania Objective-C oraz Swift [9-10]. Pozostałe z badanych bibliotek, czyli biblioteki FBLPromises oraz Promises zostały stworzone przez społeczność open-source i udostępnione publicznie przez repozytorium na stronie GitHub [11].

3.1. Biblioteki napisane w języku Objective-C

Pierwszą grupę z przygotowanych bibliotek stanowią biblioteki napisane w języku programowania Objective-C. Jest to język programowania który w latach 2007-2014 był jedynym językiem wykorzystywanym do tworzenia aplikacji i bibliotek dla systemu iOS. Mimo iż od 2014 roku zastąpił go język Swift, autorzy bibliotek wciąż mają możliwość wykorzystywania go w tworzonych bibliotekach [12].

Pierwsza z przygotowanych bibliotek jest biblioteką składającą się wyłącznie z podstawowego kodu źródłowego. Jest to zestaw pliku nagłówkowego opisującego publiczny interfejs tworzonej klasy oraz pliku źródłowego implementującego funkcje wchodzące w skład niniejszej klasy. Biblioteka ta pozwoli na stwierdzenie, czy przygotowane narzędzia będą w stanie skompilować najprostszą możliwą bibliotekę.

Kolejna biblioteka stanowi rozbudowanie poprzedniej o dodatkowe klasy opisane w nowych plikach nagłówkowych i źródłowych. Celem przygotowania niniejszej biblioteki jest zweryfikowanie jaki wpływ na proces kompilacji ma dołączanie kolejnych plików do kodu biblioteki oraz weryfikacja czy wykorzystywanie większej liczby plików uniemożliwi przeprowadzenie procesu kompilacji na systemie Linux.

Trzecią z przygotowanych bibliotek jest biblioteka, która poza kodem źródłowym wykorzystuje także zasoby graficzne. Istotne jest zweryfikowanie czy wykorzystanie takich zasobów w bibliotekach w jakikolwiek sposób wpłynie na proces ich kompilacji na systemie Linux. Na podobnej zasadzie przygotowane zostały dwie kolejne biblioteki, które odpowiednio wykorzystują katalogi zasobów w formacie xcassets oraz pliki widoków w formacie xib. Katalogi xcassets mogą w sobie zawierać zasoby różnych typów jak pliki graficzne i zestawy kolorów dla elementów graficznych aplikacji, które mogą być przygotowane w kilku wersjach dostosowanych pod różne właściwości urządzeń jak rozmiar wyświetlacza bądź aktywny motyw interfejsu [13]. Pliki xib są plikami XML i wykorzystywane są do opisu graficznego interfejsu użytkownika aplikacji [14].

Ostatnia biblioteka napisana w języku Objective-C została przygotowana w taki sposób, aby w udostępnianej przez siebie funkcjonalności wykorzystywała jedną z poprzednio przygotowanych bibliotek jako zależność dynamiczną. Umożliwi to zweryfikowanie w jaki sposób biblioteki mogą wykorzystywać kod innych bibliotek i jak wpłynie to na sposób kompilacji.

3.2. Biblioteki napisane w języku Swift

W przypadku języka programowania Swift przygotowane zostały jedynie dwie autorskie biblioteki. Pierwsza z nich składa się z minimalnego zestawu elementów, czyli jednego pliku źródłowego niewykorzystującego żadnych dodatkowych zależności. Druga zaś została rozbudowana o wykorzystanie systemowych modułów Foundation (zestaw bazowych funkcjonalności systemowych), Dispatch (zestaw wspierający tworzenie wielowątkowego kodu) oraz UIKit (zestaw do budowy i obsługi interfejsu graficznego użytkownika) [15-17]. Zbadanie czy wykorzystanie tych modułów nie zablokuje możliwości kompilacji bibliotek na systemie Linux jest istotne ze względu na fakt, iż większość z obecnie tworzonych bibliotek wykorzystuje przynajmniej jeden z nich.

Biblioteki wykorzystujące zasoby plikowe, katalogi zasobów bądź pliki xib nie zostały przygotowane w tym języku programowania, ponieważ ich kompilacja dla bibliotek napisanych w języku Swift przebiega w taki sam sposób jak dla bibliotek napisanych w języku Objective-C.

3.3. Dodatkowe biblioteki

Innym zestawem analizowanych bibliotek są biblioteki przygotowane przez społeczność open-source. Są to biblioteki Promises oraz FBLPromises z repozytorium promises stworzonego przez firmę Google [18]. Zawierają one kod ułatwiający wykonywanie asynchronicznych operacji na kilku wątkach. Pierwsza z bibliotek została napisana w całości w języku programowania Objective-C. Druga zaś została napisana w języku Swift oraz dodatkowo jako zależność wykorzystuje kod z biblioteki napisanej w języku Objective-C. Kompilacja niniejszych bibliotek pozwoli stwierdzić, czy cały proces działa poprawnie nawet w przypadku bibliotek o bardziej rozbudowanym kodzie.

Ostatnią analizowaną biblioteką jest biblioteka, której kod źródłowy napisany został zarówno w języku Objective-C oraz języku Swift. Działa ona na zasadzie udostępnienia klas, które w swojej implementacji zawierają funkcje odwołujące się do klas napisanych w drugim języku. Biblioteka ta pozwoli ona na weryfikację czy biblioteki, które wykorzystują oba języki programowania w ramach jednego kodu także będzie można poprawnie skompilować na systemie Linux.

4. Metoda badań

Analizie podlegało badanie możliwości kompilacji bibliotek na systemie Linux, czasu ich kompilacji, rozmiaru plików końcowych oraz wydajności przygotowanych bibliotek iOS na systemie operacyjnym Linux i macOS. Wszystkie z bibliotek skompilowane zostały w wersjach dostosowanych do działania na urządzeniach fizycznych opartych o architekturę arm64. Na systemie macOS każda z bibliotek skompilowana została z wykorzystaniem narzędzia xcodebuild w trybie „Debug" (konfiguracja domyślna) oraz w trybie „Release" (konfiguracja z dodatkowymi optymalizacjami). Dodatkowo wszystkie te biblioteki zostały skompilowane przy użyciu specjalnie przygotowanego skryptu bash bezpośrednio wykorzystującego niskopoziomowe narzędzia kompilujące.

Na systemie operacyjnym Linux wszystkie z przygotowanych bibliotek skompilowane zostały wyłącznie przy użyciu skryptu bash ze względu na brak narzędzia xcodebuild na tym systemie. Wszystkie z wymienionych narzędzi zostały wywołane w powłoce konsoli w ramach systemowego polecenia time pozwalającego mierzyć czas ich trwania.

Do przeprowadzenia kompilacji na systemie macOS wykorzystany został komputer o następującej specyfikacji:

- System operacyjny macOS 11.0.1
- Procesor 3,2 GHz 6-Core Intel Core i7-8700B
- Pamięć 16GB
- Karta graficzna Intel UHD Graphics 630 1536 MB

Wykorzystany komputer z systemem Linux posiadał następującą specyfikację:

- System operacyjny Ubuntu 20.04.1 LTS
- Procesor 3,2 GHz 4-Core Intel Core i5-4460
- Pamięć 8GB
- Karta graficzna Nvidia GeForce GTX 950

4.1. Sposób analizy możliwości kompilacji

Proces kompilacji przeprowadzony na systemie macOS został wykonany w kilku podejściach. Przy każdym podejściu wykorzystane zostały oddzielne narzędzia i konfiguracje. W ten sposób stwierdzono poprawność działania przygotowanych skryptów oraz umożliwiono dokładne porównanie procesu względem tego, który został wykonany na systemie Linux.

Podczas pierwszego podejścia każdą z przygotowanych bibliotek skompilowano przy użyciu narzędzia xcodebuild w konfiguracji Release. W drugim podejściu wykorzystano polecenie xcodebuild w konfiguracji Debug. Trzecie podejście do procesu kompilacji zostało przygotowane z użyciem autorskich skryptów kompilacyjnych omijających narzędzie xcodebuild. Skrypty te przeprowadzają kompilację poprzez wywoływanie bardziej niskopoziomowych od polecenia xcodebuild poleceń kompilatorów clang oraz swift. Przykładem takiego skryptu jest kod widoczny na Listingu 1.

Listing 1: Skrypt kompilujący bibliotekę Objective-C na systemie macOS

```
#!/bin/bash
set -e
FRAMEWORK_NAME="Minimal_Objective_C"
WORKING_DIR="$(pwd)"
BUILD_DIR="${WORKING_DIR}/build/Device"
FRAMEWORK_DIR="${BUILD_DIR}/${FRAMEWORK_NAME}.framework"
IOS_SDK="/Applications/Xcode
    .app/Contents/Developer/Platforms/iPhoneOS
    .platform/Developer/SDKs/iPhoneOS14.4.sdk"
rm -rf "${BUILD_DIR}"

# Krok 1. Przygotowanie struktury biblioteki
mkdir -p "${FRAMEWORK_DIR}/Headers"
mkdir -p "${FRAMEWORK_DIR}/Modules"
cp "Sources"/*.h "${FRAMEWORK_DIR}/Headers/"
cp "module.modulemap" "${FRAMEWORK_DIR}/Modules/"
cp "Info.plist" "${FRAMEWORK_DIR}/"

# Krok 2. Kompilacja pliku źródłowego
clang -x objective-c \
    -fmodules \
    -fobjc-arc \
    -fmodule-name=${FRAMEWORK_NAME} \
    -target arm64-apple-ios12.3 \
    -isysroot ${IOS_SDK} \
    -c "Sources/MOCLibraryInfo.m" \
    -o "${BUILD_DIR}/MOCLibraryInfo.o"
echo "${BUILD_DIR}/MOCLibraryInfo.o" >
    "${BUILD_DIR}/LinkFileList"

# Krok 3. Linkowanie biblioteki
xcrun -sdk iphoneos libtool -dynamic \
    @"${BUILD_DIR}/LinkFileList" \
    -install_name
        @rpath/${FRAMEWORK_NAME}
        .framework/${FRAMEWORK_NAME} \
    -arch_only arm64 \
    -o "${FRAMEWORK_DIR}/${FRAMEWORK_NAME}" \
    -syslibroot "${IOS_SDK}" \
    -lSystem \
    -compatibility_version 1 \
    -current_version 1
```

Proces kompilacji bibliotek na systemie Linux przeprowadzony został wyłącznie w jednym podejściu dla każdej biblioteki. Do kompilacji wykorzystane zostały autorskie skrypty przygotowane pod system macOS, do których wprowadzone zostały konieczne modyfikacje polegające na zastąpieniu poleceń kompilatorów clang oraz swift ich odpowiednikami w wersjach działających na systemie Linux. Takie wersje narzędzi przygotowane zostały poprzez pobranie iOS SDK, a następnie kompilację narzędzi z zestawu cctools-port i kompilatora Swift w taki sposób, aby wykorzystywały pobrany iOS SDK.

Listing 2: Skrypt analizujący czas kompilacji

```
#!/bin/bash

LIBRARY_DIR="$1"

pushd "$LIBRARY_DIR"

for i in {1..10}
do
    [ -d build ] && rm -r build
    (time ./build_device.sh > /dev/null) 2>&1 | grep
        real | awk '{print $2}'
done

popd
```

Podczas wykonywania wszystkich operacji kompilacji mierzony był czas ich działania za pośrednictwem specjalnie przygotowanego skryptu zaprezentowanego

na Listingu 2. Działa on na zasadzie uruchomienia polecenia kompilacyjnego 10 razy na podanej bibliotece i wyświetlenia czasu trwania poszczególnych operacji kompilacji w sekundach.

4.2. Sposób analizy rozmiaru bibliotek

Analiza rozmiaru bibliotek przeprowadzona została poprzez uruchomienie przygotowanego skryptu wyświetlającego rozmiary w bajtach bibliotek znajdujących się we wskazanym katalogu.

Listing 3: Skrypt analizujący rozmiar bibliotek

```
#!/bin/bash

DIRECTORY="$1"

for framework in $(ls $DIRECTORY)
do
    echo -n "$framework - "
    find "$DIRECTORY/$framework" | xargs stat -f%z | awk
        '{ s+=$1 } END { print s }'
done
```

Skrypt ten (Listing 3) działa na zasadzie odwołania się do systemowych poleceń w celu odczytania informacji o rozmiarze całej biblioteki, czyli licząc rozmiar pliku wykonywalnego biblioteki jak i pozostałych plików znajdujących się w jej strukturze. Jako parametr wejściowy przekazany został katalog, w którym umieszczone zostały wszystkie z przygotowanych bibliotek. Dzięki temu otrzymano informacje o rozmiarach każdej biblioteki wykonując tylko jedno wywołanie niniejszego skryptu.

4.3. Sposób analizy wydajności bibliotek

Analiza wydajności przygotowanych bibliotek przeprowadzona została poprzez zbadanie wpływu użycia danej biblioteki na czas uruchamiania aplikacji testowej oraz zbadanie czasu wykonywania funkcji udostępnianych przez wszystkie z bibliotek.

Do pomiaru wpływu bibliotek na czas startu aplikacji wykorzystana została zmienna środowiskowa „DYLD_PRINT_STATISTICS”, która aktywuje wbudowaną funkcjonalność pomiarową polecenia dyld odpowiedzialnego za operację wczytywania bibliotek dynamicznych [19]. Zmienna ta została aktywowana w momencie uruchamiania przygotowanej aplikacji testowej. Czas zwracany przez tę funkcjonalność oznacza jak długo trwało ładowanie aplikacji do wywołania jej głównej funkcji „main()”. W wyniku zwracany jest zestaw czasów (Rysunek 1), z których interesującym nas wpisem jest wpis „dylib loading time”, który bezpośrednio dotyczy czasu wczytywania bibliotek dynamicznych.

```
Total pre-main time: 222.72 milliseconds (100.0%)
         dylib loading time: 142.73 milliseconds (64.0%)
        rebase/binding time:  25.28 milliseconds (11.3%)
            ObjC setup time:   5.91 milliseconds (2.6%)
           initializer time:  48.78 milliseconds (21.9%)
           slowest intializers :
             libSystem.B.dylib :   5.75 milliseconds (2.5%)
   libBacktraceRecording.dylib :  10.48 milliseconds (4.7%)
    libMainThreadChecker.dylib :  28.91 milliseconds (12.9%)
```

Rysunek 1: Przykładowy wynik czasów trwania poszczególnych operacji wykonywanych przy starcie aplikacji

Drugim analizowanym aspektem wydajności bibliotek był pomiar czasu trwania funkcji udostępnianych przez wszystkie biblioteki. W tym celu przygotowany został test jednostkowy. Listing 4 prezentuje jak niniejszy test odwołuje się do funkcji każdej biblioteki w ramach bloku measure. Napisany został z wykorzystaniem biblioteki XCTest będącej systemową biblioteką pozwalającą na pisanie testów jednostkowych z możliwością pomiaru wydajności [20].

Listing 4: Test jednostkowy wykonujący pomiar wydajności bibliotek

```
class TestAppTests: XCTestCase {
    func testPerformance() throws {
        measure {
            _ = MOCLibraryInfo().name
            _ = MOCLibraryInfo().language
            EOCFile1.description()
            EOCFile2.description()
            EOCFile3.description()
            EOCFile4.description()
            EOCFile5.description()
            AOCProvider.provideTextFileContent()
            AOCProvider.provideImage()
            DOCUseDependency.textFromDependency()
            OACReadAssets.readColorFromAssets()
            OACReadAssets.readImageFromAssets()
            OVDemoView.fromNib()
            _ = SwiftLibraryInfo.name
            _ = SwiftLibraryInfo.language
            _ = SwiftFoundationWorker.makeFoundationObject()
            _ = SwiftFoundationWorker.makeUILabel()
            SwiftFoundationWorker.doDelayedWork()
            Promise { "Text" }
                .delay(2)
                .then { print("Swift promises \($0()) work!") }
            MLObjcFile.getSwiftText()
            _ = SwiftFile().getObjcText()
        }
    }
}
```

5. Wyniki badań

5.1. Możliwość kompilacji

Podstawowym celem niniejszej pracy było określenie z jakich zależności bądź funkcjonalności mogą korzystać biblioteki, aby możliwa była ich kompilacja za pośrednictwem systemu Linux. Tabela 1 przedstawia zestawienie poszczególnych elementów z jakich mogą składać się biblioteki w zależności od tego w jakim języku napisana została analizowana biblioteka.

Tabela 1: Wyniki kompilacji bibliotek na systemie Linux

	Swift	Objective-C
Kompilacja możliwa	tak	tak
Systemowe moduły (np. Foundation, UIKit)	tak	tak
Zasoby plikowe (.txt, .png, .jpg)	tak	tak
Katalogi zasobów (.xcassets)	nie	nie
Zasoby widoków (.xib)	nie	nie
Zależności bibliotek	tak	tak

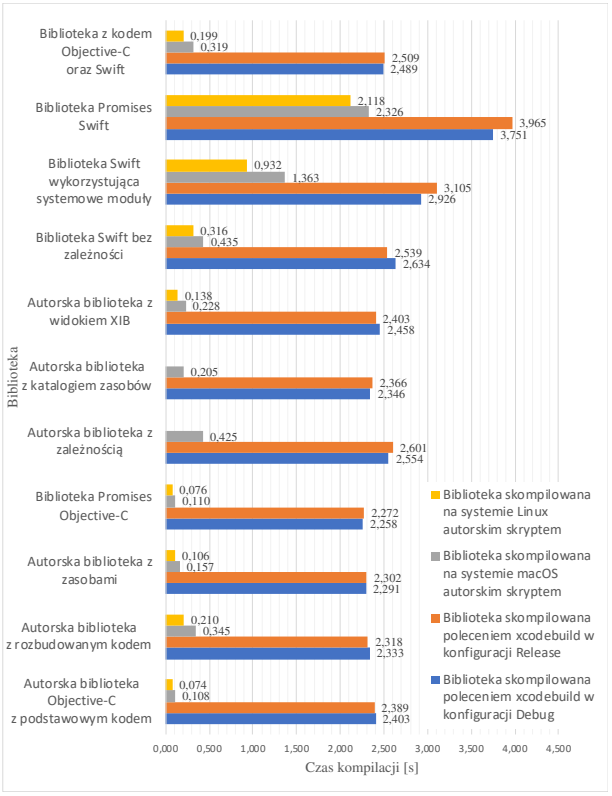
Tabela ta pokazuje, które z bibliotek udało się poprawnie skompilować na danych systemach, a które nie. Stwierdzenie „tak” oznacza, że biblioteka została skompilowana oraz poprawnie uruchomiona w testowej aplikacji. Stwierdzenie „nie” oznacza, że nie udało się znaleźć istniejących narzędzi open-source, które pozwoli-

łyby na przeprowadzenie kompilacji na systemie Linux, w związku z czym stwierdzono, że obecnie taka kompilacja nie jest możliwa. W przypadku systemu macOS będącego oficjalnym systemem do tworzenia bibliotek iOS takie zestawienie nie było konieczne, ponieważ kompilacja udała się z powodzeniem w każdej możliwej konfiguracji.

Główną właściwość jaką można zauważyć w tabeli 1 jest fakt, iż większość funkcjonalności nie blokuje możliwości ich kompilacji na systemie Linux. Jedynie dla bibliotek zawierających katalogi zasobów w postaci xcassets bądź zasoby widoków w postaci plików xib nie udało się pomyślnie przeprowadzić pełnej kompilacji. Kompilacja ta nie udała się z powodu braku odpowiednich narzędzi, które byłyby w stanie przetworzyć podane pliki w postać końcową rozumianą przez aplikacje iOS.

5.2. Czasy kompilacji

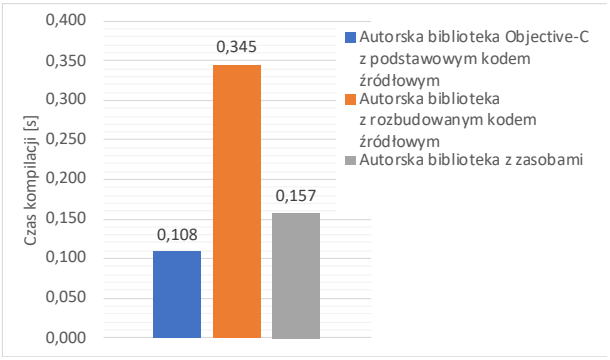
Podczas analizy czasu kompilacji bibliotek pierwszą interesującą cechą jaką zaobserwowano są znacznie dłuższe czasy kompilacji przeprowadzonych z wykorzystaniem polecenia xcodebuild w konfiguracjach Debug lub Release w porównaniu do czasów kompilacji wykonanych z wykorzystaniem autorskiego skryptu. Dla każdej biblioteki czasy kompilacji tymi narzędziami były dłuższe niż 2 sekundy, co widoczne jest na rysunku 2. W odróżnieniu do tego proces kompilacji przeprowadzony z wykorzystaniem autorskich skryptów był krótszy i osiągnął maksymalny czas 2 sekund tylko w przypadku biblioteki Promises napisanej w języku Swift.



Rysunek 2: Średnie czasy kompilacji bibliotek.

Porównując czasy kompilacji różnych bibliotek, które zostały skompilowane autorskim skryptem na systemie macOS, co zostało zaprezentowane na rysunku 3, widać, że dodając kolejne pliki bądź dodatkowe zasoby wydłużamy czas kompilacji.

Jest to spowodowane tym, że dla każdego takiego pliku wywoływane są dodatkowe operacje, które wydłużają cały proces. Widzimy także, że czas kompilacji biblioteki o rozbudowanym kodzie źródłowym jest dłuższy niż czas kompilacji biblioteki z jednym plikiem i z dołączonymi zasobami. Taka różnica wynika z tego, że pliki graficznie nie są poddawane zamianie na pliki wykonywalne, a jedynie są kopiowane do odpowiedniego miejsca w wynikowym folderze biblioteki. Pliki źródłowe zaś muszą zostać poddane wymienionemu procesowi kompilacji, który jest bardziej czasochłonny od zwykłej operacji kopiowania.



Rysunek 3: Porównanie czasów kompilacji różnych bibliotek autorskim skryptem na systemie macOS.

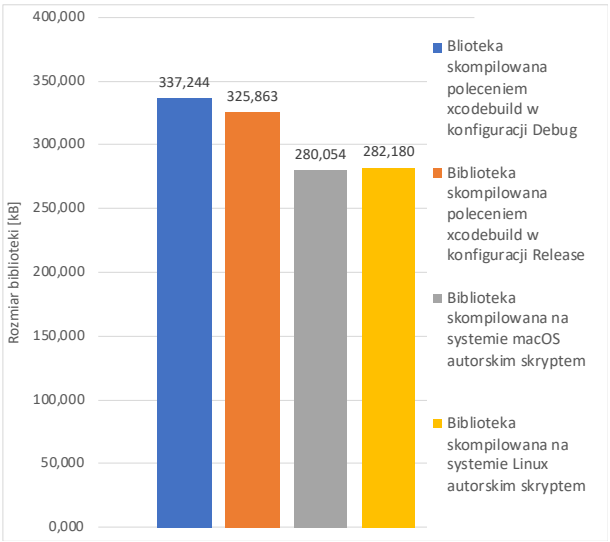
5.3. Rozmiar bibliotek

Kolejnym analizowanym aspektem jest rozmiar skompilowanych bibliotek. Zestawienie wyników dla wszystkich z przygotowanych bibliotek skompilowanych na systemie macOS i Linux przy użyciu różnych sposobów kompilacji zaprezentowano w tabeli 2. Widoczną cechą jest znacznie większy rozmiar bibliotek wykorzystujących zasoby bądź katalog zasobów w porównaniu do pozostałych bibliotek. Wynika to z faktu, iż takie zasoby dołączone są bezpośrednio do struktury bibliotek tym samym zwiększając ich rozmiary. Analizowane biblioteki, które wykorzystywały takie zasoby zawierają w sobie plik graficzny o rozmiarze około 1,5 MB, dlatego też ich rozmiar jest większy od pozostałych bibliotek o taką wartość (pomijając rozmiar samego pliku wykonywalnego).

Zestawienie uśrednionych rozmiarów skompilowanych bibliotek pod względem sposobu ich kompilacji przedstawiono na rysunku 4. Widoczną różnicą jest mniejszy rozmiar bibliotek skompilowanych przy użyciu autorskiego skryptu w porównaniu do bibliotek zbudowanych przy użyciu poleceń xcodebuild. Różnice te wynikają z zestawu operacji wykonywanych w ramach polecenia xcodebuild, które nie są wykonywane podczas kompilacji z użyciem autorskiego skryptu, a które wpływają na rozmiar utworzonego pliku wykonywalnego.

Tabela 2: Rozmiary skompilowanych bibliotek

Biblioteka	Rozmiar biblioteki skompilowanej poleceniem xcodebuild w konfiguracji Debug [kB]	Rozmiar biblioteki skompilowanej poleceniem xcodebuild w konfiguracji Release [kB]	Rozmiar biblioteki skompilowanej na systemie macOS autorskim skryptem [kB]	Rozmiar biblioteki skompilowanej na systemie Linux autorskim skryptem [kB]
Biblioteka z podstawowym kodem źródłowym	92,24	92,016	52,4	53,584
Biblioteka z rozbudowanym kodem źródłowym	80,614	80,502	54,434	55,626
Biblioteka z zależnością	75,187	75,203	58,226	69,758
Biblioteka z katalogiem zasobów	1604,568	1604,584	1581,143	-
Biblioteka z zasobami	1591,421	1591,405	1573,997	1569,001
Biblioteka z widokiem XIB	80,17	80,106	56,423	-
Biblioteka „FBLPromises”	301,498	268,842	222,36	234,22
Biblioteka „Promises”	501,555	436,483	306,572	302,811
Biblioteka Swift bez zależności	126,64	125,48	80,996	82,099
Biblioteka Swift wykorzystująca systemowe moduły	134,971	132,283	87,176	87,035
Biblioteka z kodem Objective-C oraz Swift	131,072	130,552	84,328	85,487



Rysunek 4: Uśrednione rozmiary bibliotek skompilowanych różnymi sposobami.

Przykładem takiej operacji jest kompilacja oraz dołączanie dodatkowego pliku uzupełniającego kompilowaną bibliotekę o dodatkowe symbole określające informacje o jej wersji. Generowanie i kompilacja niniejszego pliku została pominięta przy tworzeniu autorskiego skryptu, ponieważ nie jest on konieczny do prawidłowego działania biblioteki, a przygotowanie odpowiednich poleceń komplikowałoby dodatkowo cały proces kompilacji na obu systemach.

5.4. Wydajność bibliotek

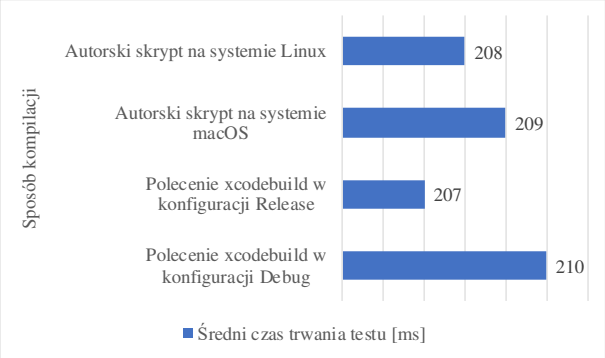
Przy analizie wydajności zbudowanych bibliotek głównym badanym elementem był ich wpływ na czas uruchamiania aplikacji testowej. Tabela 3 pokazuje otrzymane wyniki w rozbiciu na sposób kompilacji bibliotek. Dla bibliotek skompilowanych autorskim skryptem wpływ na czas ich wczytywania przy starcie aplikacji jest podobny bez względu na to, na którym systemie zostały skompilowane. Na tej podstawie możemy stwierdzić, że narzędzia, które zostały przygotowane do kompilacji bibliotek na systemie Linux nie wpłynęły negatywnie na sposób ich działania podczas startu aplikacji.

Większa różnica występuje, jeżeli porównamy biblioteki skompilowane z wykorzystaniem autorskiego skryptu do bibliotek skompilowanych z wykorzystaniem narzędzia xcodebuild, których wczytywanie trwało dłużej średnio o 7-12 ms. Różnica ta wynika głównie z większego rozmiaru bibliotek skompilowanych za pośrednictwem narzędzia xcodebuild.

Drugim mierzonym aspektem wydajności bibliotek był pomiar czasu odwoływania się do funkcji udostępnianych przez publiczny interfejs skompilowanych bibliotek. Uśrednione wyniki takich pomiarów zaprezentowane zostały na rysunku 5. Wszystkie z otrzymanych wyników mieszczą się w przedziale od 207 ms do 210 ms.

Tabela 3: Wyniki czasów wczytywania bibliotek przy starcie aplikacji

Sposób kompilacji	Wartość minimalna [ms]	Wartość maksymalna [ms]	Średnia [ms]	Mediana [ms]
Polecenie xcodebuild w konfiguracji Debug	143,56	149,22	146,97	147,19
Polecenie xcodebuild w konfiguracji Release	144,89	152,23	148,73	152,23
Autorski skrypt na systemie macOS	137,42	143,09	140,66	140,94
Autorski skrypt na systemie Linux	136,97	142,73	139,64	139,18



Rysunek 5: Porównanie wydajności skompilowanych bibliotek.

Takie wyniki oznaczają, że sposób kompilacji nie wpłynął negatywnie na wydajność kodu bibliotek. Zarówno biblioteki, które zostały skompilowane na systemie macOS, jak i te, które zostały skompilowane na systemie Linux działają w ten sam sposób. Jest to spowodowane prostym kodem przygotowanych bibliotek, na którym narzędzia kompilacyjne nie miały możliwości przeprowadzenia operacji optymalizacyjnych.

6. Wnioski i podsumowanie

Niniejszy artykuł opisuje analizę procesu kompilacji bibliotek iOS na systemie Linux. W badaniu w pierwszej kolejności zweryfikowano, czy kompilacja bibliotek iOS jest możliwa na tym systemie dla podstawowych typów bibliotek. Następnie sprawdzono, czy istnieją funkcjonalności wykorzystywane w bibliotekach, które uniemożliwią wykonanie kompilacji na systemie Linux. Ostatecznie zbadano: czas kompilacji, rozmiary oraz wydajność skompilowanych bibliotek.

Wyniki uzyskane w niniejszej pracy pokazały, że kompilacja bibliotek iOS możliwa jest do przeprowadzenia na systemie Linux pod warunkiem, że nie będą one korzystać z niektórych funkcjonalności. Zidentyfikowano, że wykorzystanie katalogów zasobów o rozszerzeniu xcassets bądź plików opisujących widoki interfejsu użytkownika tworzone w postaci plików w formacie xib nie pozwalają na przeprowadzenie kompilacji bibliotek na systemie Linux.

Analiza czasów kompilacji pokazała, że wykorzystując system Linux możliwe jest uzyskanie zbliżonych, a nawet krótszych czasów kompilacji bibliotek w porównaniu do systemu macOS. Niestety dzieje się to kosztem rozmiaru bibliotek, które na podstawie przeprowadzonej analizy są większe w przypadku bibliotek skompilowanych wykorzystując narzędzia na systemie Linux. Ostatnia z analiz, czyli analiza wydajności bibliotek pokazała, że zarówno biblioteki skompilowane na systemie macOS jak i biblioteki skompilowane na systemie Linux mają zbliżoną do siebie wydajność.

Literatura

[1] What are Frameworks?, https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html#//apple_ref/doc/uid/20002303-BBCEIJFI, [03.2021].

[2] S. Grimshaw, Mastering MacOS Programming, Packt Publishin, 2017.

[3] B. Alexander, J. B. Dillon, K. Y. Kim, R. Górczyński, Tworzenie aplikacji na platformę iOS 5: z wykorzystaniem Xcode, Interface Builder, Instruments, GDB oraz innych kluczowych narzędzi, Wydawnictwo Helion, 2012.

[4] R. Pouclet, Pro iOS Continuous Integration, Apress 2014.

[5] Godot Engine – Free and open source 2D and 3D game engine, https://godotengine.org, [03.2021].

[6] Cross-compiling for iOS on Linux – Godot Engine latest documentation, https://docs.huihoo.com/godotengine/godot-docs/godot/reference/cross-compiling_for_ios_on_linux.html, [03.2021].

[7] Apple cctools port for Linux and *BSD, https://github.com/tpoechtrager/cctools-port, [03.2021].

[8] Automated embedding of dynamic libraries into iOS applications from GNU/Linux, https://docplayer.net/60535186-Automated-embedding-of-dynamic-libraries-into-ios-applications-from-gnu-linux.html, [03.2021].

[9] S. G. Kochan, Ł. Piwkom, Objective-C: praktyczny podręcznik tworzenia aplikacji na systemy iOS i Mac OS X!, Helion, 2012.

[10] P. Pasternak, Swift od podstaw: praktyczny przewodnik, Helion, 2017.

[11] A. Pipinellis, GitHub Essentials, Packt Publishing, 2015.

[12] C. G. Garcia, J. P. Espada, B. C. Pelayo G-Bustelo, J. M. Cueva Lovelle, Swift vs. Objective-C: A New Programming Language, International Journal of Interactive Multimedia and Artificial Intelligence 3(3) (2015) 74-81, http://dx.doi.org/10.9781/ijimai.2015.3310.

[13] Asset Catalog Format Reference, https://developer.apple.com/library/archive/documentation/Xcode/Reference/xcode_ref-Asset_Catalog_Format/index.html, [03.2021].

[14] Xcode Overview: Using Interface Builder, https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingInterfaceBuilder.html, [03.2021].

[15] Foundation – Apple Developer Documentation, https://developer.apple.com/documentation/foundation, [03.2021].

[16] Dispatch – Apple Developer Documentation, https://developer.apple.com/documentation/dispatch, [03.2021].

[17] UIKit – Apple Developer Documentation, https://developer.apple.com/documentation/uikit, [03.2021].

[18] Promises is a modern framework that provides a synchronization construct for Swift and Objective-C, https://github.com/google/promises, [03.2021].

[19] Man page dla polecenia dyld, https://www.manpagez.com/man/1/dyld/, [03.2021].

[20] N. Godfrey, Agile Swift: Swift Programming Using Agile Tools and Techniques, Apress, 2016.