学 士 学 位 論 文

題　　目

AArch64 アーキテクチャにおけるアンワインド情報検査／合成システムの開発

Development of Validation / Synthesis System for Stack Unwinding Information on AArch64

指 導 教 員

宮地　充子

報 告 者

川口　哲弘

令和 4 年 2 月 7 日

大阪大学工学部　電子情報工学科

概 要

高級言語で実装されたプログラム実行時のスタックアンワインドは，DWARF で記述されたデバッグ情報に基づいてなされている．この情報は，Theseus というメモリ安全性に重きを置く OS にも利用されており，障害回復の一端を担っている．しかし，デバッグ情報にはバグが存在するという報告がされており，信頼性が失われている．この問題に対して，x86_64 アーキテクチャコンピュータ用のデバッグ情報の検査システムや合成システムが提案されている．x86_64 用システムは，命令セットや呼び出し規約の異なる AArch64 アーキテクチャに対応できないため，本論文では，x86_64 アーキテクチャと AArch64 アーキテクチャの違いを踏まえた AArch64 アーキテクチャ用のデバッグ情報検査，合成システムを提案する．

Abstract

The stack unwinding during program execution implemented in high-level languages is based on debugging information written in DWARF. This information is used by Theseus, an operating system that emphasizes memory safety, and plays a role in fault recovery. However, it has been reported to contain bugs, which makes it unreliable. To address this problem, validation and synthesis system of debugging information for x86_64 has been proposed. It cannot support the AArch64 architecture bacause it has a different instruction set and calling convention from x86_64. In this paper, we propose validation and synthesis system of debugging information for the AArch64 architecture based on the differences between the x86_64 and AArch64 architectures.

目 次

第1章　はじめに

1.1　研究背景

スマートフォンや組み込み機器においてARM製CPUが広く使われている．サーバやPC向けに64ビット拡張されたものをAArch64アーキテクチャと呼び，その電力効率の良さからIntel64やAMD64といったx86_64アーキテクチャCPUに取って代わる製品も見られる．こうしたAArch64アーキテクチャCPUの普及拡大によって，その上で動かす安全性が確保されたシステムの開発も急がれる．こうした中，x86_64アーキテクチャ用にTheseusというOSがKevin Boosらによって開発された [3]．TheseusはRustの言語レベルの機構によってメモリ安全性や各プロセスの独立性が担保されており，安全性の要として，DWARF形式のアンワインド情報が使用されている．この安全なシステム設計をAArch64アーキテクチャに適用させることが我々の将来的な目標である．しかし，安全性確保のためのアンワインド情報はバグを含んでいることが報告されている [10]．そのため，まずアンワインド情報のバグの検出を行い，その部分の正しいアンワインド情報を合成することが求められる．

1.2　本研究の目的と貢献

既存研究として．x86_64アーキテクチャ用のアンワインド情報検査システムと合成システム [2] が提案されているが，x86_64アーキテクチャとAArch64アーキテクチャでは命令セットや呼び出し規約などで大きな違いがあるため，x86_64用のアルゴリズムをそのままAArch64アーキテクチャに適用することはできない．本研究では，これら二つのアーキテクチャの違いを明らかにし，AArch64アーキテクチャに適した検査システムと合成システムの設計，開発を目的とする．

本研究の貢献は検査システムと合成システムの設計にある．

検査システム

検査システムにおける貢献は，既存研究の設計方針を拡張したことである．図1.1の通り，既存研究の設計方針である，アンワインド情報におけるスタックポインタ更新の記述通りに，実際の検査対象プログラムが実行されていることを検査する機能を拡張し，callee-savedレジスタのスタック保存情報通りに，実際の検査対象プログラムが実行されていることを検査する機能（**機能1**）とした．さらに，アンワインド情報に記載されていないcallee-savedレジスタのスタック保存が実際の検査対象プログラムで実行されていないことを検査する機能（**機能2**），スタック保存されたcallee-savedレジスタの値が，関数終了時，アンワインド情報通りにレジスタ復帰していることを検査する機能（**機能3**）を追加した．なお，この設計方針はx86_64アーキテクチャの検査システムにも適用可能である．

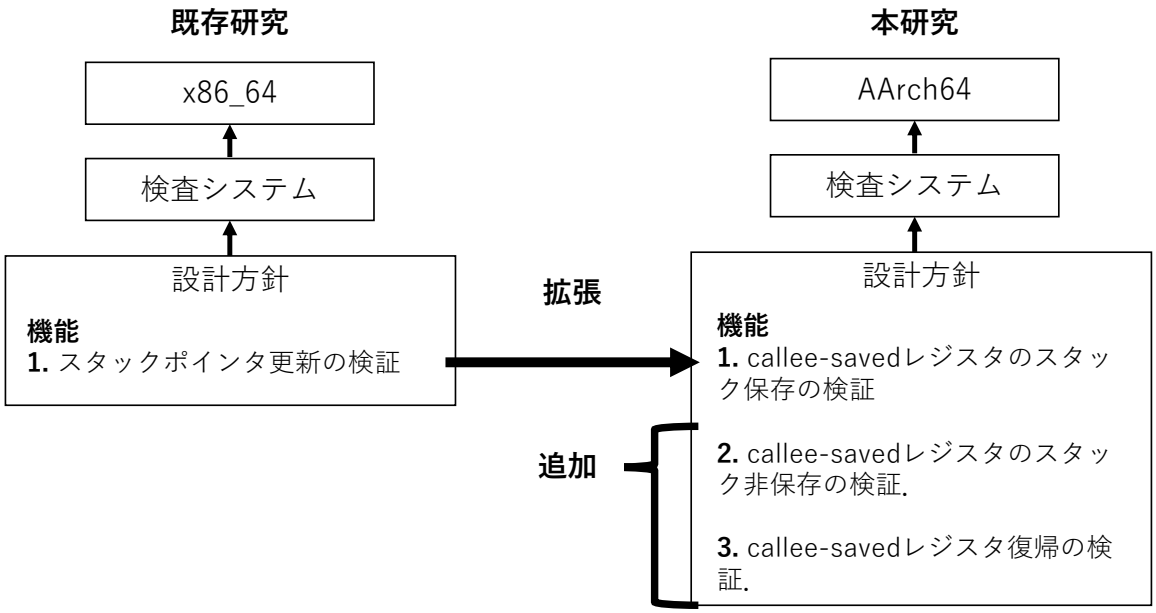

図 1.1: 検査システムにおける既存研究と本研究の違い

合成システム

合成システムにおける貢献は，既存研究のアンワインド情報合成方法を変更したことである．図 1.2 の通り，既存研究の合成システムでは，アンワインド情報の合成方法は x86_64 アーキテクチャの命令セットである push 命令に依存している．本研究で対象になる AArch64 アーキテクチャでは push 命令は存在せず，代わりに str，stp 命令が使用されている．この AArch64 の命令セットに従ったアンワインド情報合成方法を実装した.

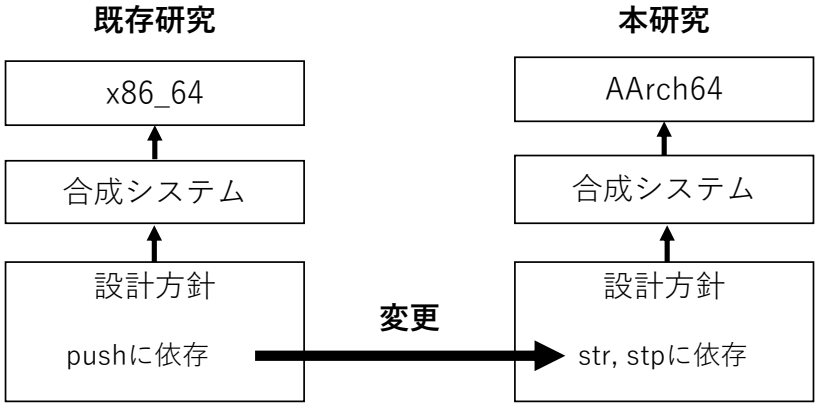

図 1.2: 合成システムにおける既存研究と本研究の違い

1.3 本論文の構成

最後に本論文の構成について記載する．2 章では，x86_64 アーキテクチャと AArch64 アーキテクチャの特徴とアンワインド情報における基礎知識をまとめる．3 章では，TheseusOS の基本的な構造と x86_64 アーキテクチャ用アンワインド情報検査・合成アルゴリズムについてまとめる．4 章では，AArch64 アーキテクチャ用アンワインド情報検査・合成システムの設計と実装について記載する．5 章では，AArch64 アーキテクチャ用アンワインド情報検査・合成システムをそれぞれ実際動かしたときの結果とそれに対する考察をまとめる．6 章では，本研究のまとめを記載する．

第2章　準備

2.1　CPU アーキテクチャ

2.1.1　x86_64 アーキテクチャ

Intel 64 や AMD 64 はどちらも IA_32 アーキテクチャという 32 ビット命令セットアーキテクチャを 64 ビット拡張したアーキテクチャであるため，それらを総称して x86_64 と呼ばれている．ここでは，Intel64 アーキテクチャについて説明をする [11]．使われる命令セットは CISC ライク，プログラウカウンタは rip レジスタ，スタックポインタは rsp レジスタ，フレームポインタは rbp レジスタが使われている．レジスタの大きさは 64 ビットであり，32，16，8 ビットでのアクセスも可能である．64 ビット汎用レジスタ rax，rbx，rcx のように 64 ビットでアクセスする場合は，名称の先頭が r になっている．32 ビット汎用レジスタを使うときは，64 ビット汎用レジスタの下位 32 ビットが利用され，eax，ebx，ecx のように名称の先頭が r から e に置き換わる．

表 2.1 は，Intel64 アーキテクチャの主な命令セットである．

表 2.1: Intel 64 の主な命令セット

命令	意味
push	データをコールスタックにプッシュする
pop	コールスタックの rsp の指すデータをポップする
add	二つの数値を加算する
mov	データをコピーする
call	関数（サブルーチン）を呼び出す
ret	現在の関数（サブルーチン）から呼び出し元に戻る

（命令末尾に q，l，w，b が付く事があるが，それぞれオペランドに対して 64，32，16，8 ビットアクセスすることを意味する．）

次に，x86_64 における GNU/Linux の Application Binary Interface (ABI） [8] について説明する．ABI には，関数呼び出しにおける規則のことである．まず，関数呼び出しの際には call 命令が実行される．この命令の実態は，この命令の同一関数内の次の命令アドレス (＝リターンアドレス) をコールスタックにプッシュし，呼び出し関数の先頭の命令を示すアドレスにプログラムカウンタを書き換えることである．呼び出し関数に引数を渡す場合は，レジスタを用い，rdi，rsi，rdx，rcx，r8，r9 をこの順で使うように定められており，引数がこれより多い場合には，コールスタック内に保存する．これは，OS やコンパイラによって異なる．呼び出された関数の最後には ret 命令が実行される様になっており，この命令は call 命令によってプッシュされたリターンアドレスをポップし，その値にプログラムカウンタを書き直すことである．これによって，呼び出し元の call 命令の次の命令を引き続いて実行することが可能である．

レジスタにはそれぞれ細かい役割が定義されているが，ABI の観点から，役割を２つに分けて説明する．１つ目は，caller-saved レジスタと呼ばれるレジスタ群である．別名，揮発性レジスタとも呼ばれ，一時的な情報を保持するレジスタである．このレジスタは，関数呼び出しが起こるプログラムにおいて，呼び出し前にそのレジスタの値をコールスタック上に退避させる必要があり，サブルーチン終了後にはコールス

タックからレジスタに値を読み出して復帰する．しかし，サブルーチン終了後に使用する必要のないレジスタはコールスタックに退避させる必要がない．したがって，関数呼び出し前後でレジスタの状態が変化している可能性がある．具体的には，rax, rcx, rdx, rsi, rdi, r8, ⋯ ,r11 などのレジスタがこれに該当する．２つ目は，callee-saved レジスタと呼ばれるレジスタ群である．別名，不揮発性レジスタと呼び，長寿命な情報を保持するレジスタである，このレジスタは，関数呼び出しが起こるプログラムにおいて，呼び出し先の実行時にコールスタックへ退避する可能性のあるレジスタである．サブルーチン終了時にコールスタックからレジスタに読み込んで復帰し，呼び出し元の関数へ実行を戻す．そのため，関数呼び出し前後でレジスタの状態が変化することはない，このレジスタは，必ずコールスタックに値を退避するわけではなく，サブルーチン内の実行でこのレジスタを書き換えてしまう恐れのある場合に限り，呼び出し先の責任で事前に退避，事後に復帰する．具体的には，rbx, rbp, r12, ⋯ ,r15 のレジスタがこれに該当する．

図 2.1.1 は，コールスタックの構造を示している．コールスタックのメモリアドレスは，スタックの先頭に行くほど小さくなる構造になっており，基本的にスタックデータの先頭アドレスを rsp が保存している．前述の通り，関数呼び出しの際にはリターンアドレスがプッシュされるのだが，呼び出し関数の最初の命令で，呼び出し元関数実行時に保存されていた rbp の値がリターンアドレスの上の領域にプッシュされる．この動作は，コンパイル時に -fomit-frame-pointer [5] のオプションによって rbp がフレームポインタとして使用されないときには行われない．続く呼び出し関数内の実行時に使用される値を保存するローカル変数領域，関数呼び出しがあった際に引数を保存する引数領域があり，コールスタックという文字通り積み上げるように変数が保存される．また，ある関数 fn があり，その関数で使用される引数領域，リターンアドレス，呼び出し元のフレームポインタの値，ローカル変数領域をまとめて関数 fn のスタックフレームと呼ぶ．

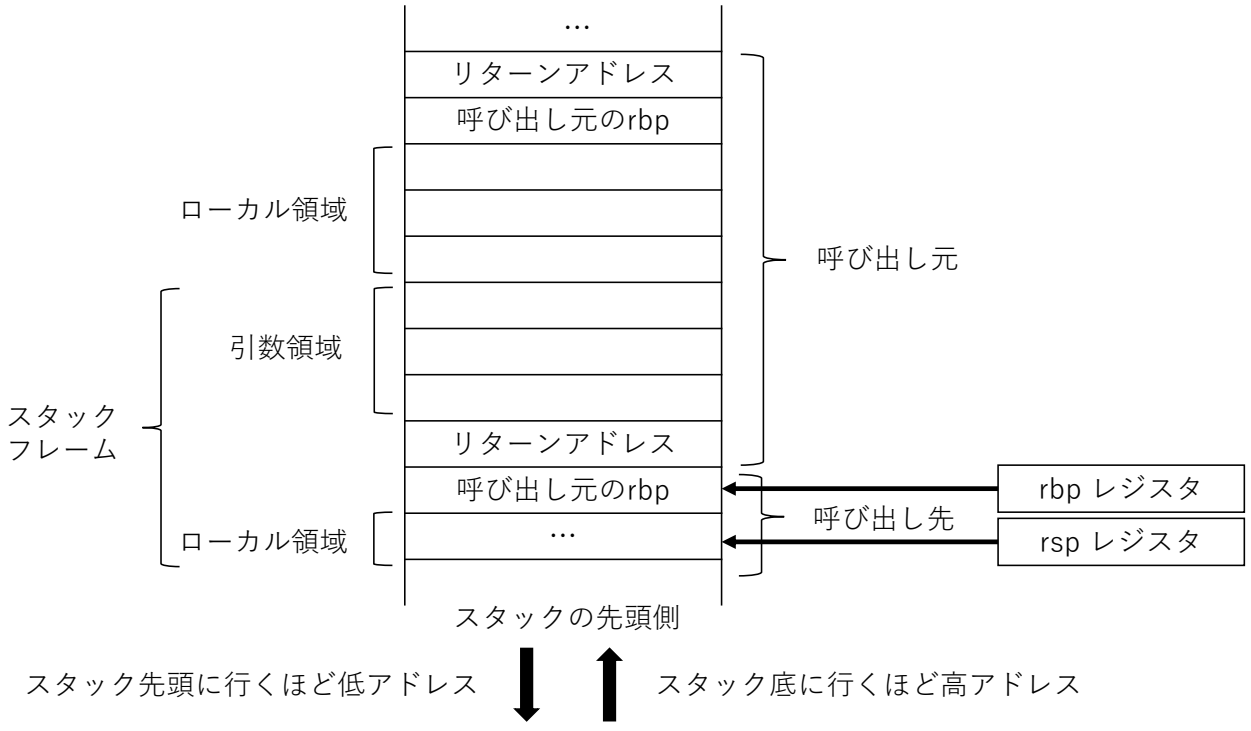

図 2.1: x86_64 におけるコールスタックの構造

2.1.2 AArch64 アーキテクチャ

AArch64 は Arm 社がプロセッサの設計を行った ARM アーキテクチャの 64 ビット拡張したものである．ARM アーキテクチャには，様々なバージョンの命令セットアーキテクチャがあり，例えば，命令セットアーキテクチャのバーション 8 が Armv8 と呼ばれる．スマートフォンの CPU に Arm の命令セットアーキテクチャが使われていることが多く，Qualcomm の Snapdragon シリーズや，Apple の A シリーズ CPU には Arm の命令セットアーキテクチャが使われている．ここでは，Armv8-A アーキテクチャについて説明する [12]．使われる命令セットは RISC ライク，プログラムカウンタは pc レジスタ，スタックポインタは sp ポインタ，フレームポインタは x29 レジスタが使われている．大きな特徴として x30 レジスタがリターンアドレスを保存するためのレジスタとして用意されている．64 ビット汎用レジスタは x0，x2，…，x30 と名前がつけられていて，一部のレジスタを除いて，それぞれのレジスタの役割は決まっていない．これらのレジスタの下位 32 ビットは 32 ビット汎用レジスタに使われ，w0，w1，…，w30 と名前が変わる．また，Armv8-A には浮動小数点数あよびベクトル演算用のレジスタが用意されており，v0，v1，…，v31 である．

表 2.2 は，Armv8-A の主な命令セットである．

表 2.2: Armv8-A の主な命令セット

命令	意味
ldr	メモリのデータをレジスタに読み込む
str	レジスタのデータをメモリに書き込む
add	二つの数値を加算する
mov	データをコピーする
bl	関数（サブルーチン）を呼び出す
ret	現在の関数（サブルーチン）から呼び出し元に戻る

x86_64 のとの大きな違いとして，push 命令や pop 命令が Armv8-A の命令セットにはない．

次に，AArch64 における GNU/Linux の ABI [1] について説明する．まず，x86_64 とは違い，関数呼び出しのアセンブリ命令は bl または blr である．この命令では，呼び出し関数の先頭の命令を示すアドレスにプログラムを書き換えるのは call 命令と同様だが，リターンアドレスをコールスタックにプッシュするのではなく，x30 レジスタに保存する．ret 命令実行時にはは x30 の値にプログラムカウンタを変更することで，呼び出し元の関数に復帰することができる．x30 レジスタの存在によって，コールスタックにはリターンアドレスを保存しないのかというと，そうではない．呼び出し先の関数内で，別の関数呼び出しがプロブラムされているときは，呼び出し関数の先頭の命令で x30 レジスタの値をコールスタックに保存する，呼び出し先の関数が終了するときには，コールスタックに保存されたリターンアドレスを x30 レジスタに復帰させて ret 命令が実行される．逆に，呼び出し先の関数で，別の関数呼び出しがされないときは，コールスタックにリターンアドレスを保存せず，x30 レジスタが書き換えられることがないため，呼び出し先関数終了時には x30 レジスタに何もすることはなく ret 命令が実行されるという動作でリターンアドレスを管理している．関数呼び出しの際の引数は汎用レジスタ，浮動小数点レジスタはそれぞれ，x0, … , x7, v0, … , v7 がこの順に使用され，引数の数がこのレジスタの数よりも多い場合，コールスタックに保存される．caller-saved レジスタと callee-saved レジスタの役割は x86_64 と同様で，AArch64 の場合，caller-save レジスタが r9, … , r15, v16, … , v31．callee-save レジスタが r19, … , r28, v8, … , v15 である，

図 2.1.2 は，AArch64 のコールスタックの構造を示している．x86_64 との大きな違いは，リターンアドレスを保存するタイミングと場所である．AArch64 ではリターンアドレスの保存を呼び出し先の関数実行中に行う．この際，単にプッシュするのではなく，事前にその関数で使用するローカル領域分の空間を確保しておいて，呼び出し元で使われるメモリ領域と間が空いた形でリターンアドレスを保存する．関数実

行中に保存されるローカル変数はスタックに積み重ねるのではなく，先程開けたメモリ領域に，sp からどれだけ離れたアドレスに保存するかということを指定して保存している，この保存方法の違いは x86_64 の push 命令と AArch64 の str 命令の違いを反映したようになっている．また，関数呼び出しの際の引数の保存は，リターンアドレス（x30 レジスタの値）が保存されているアドレスよりも小さいアドレス側に保存されるので，スタックの領域の順序が x86_64 とは異なる．

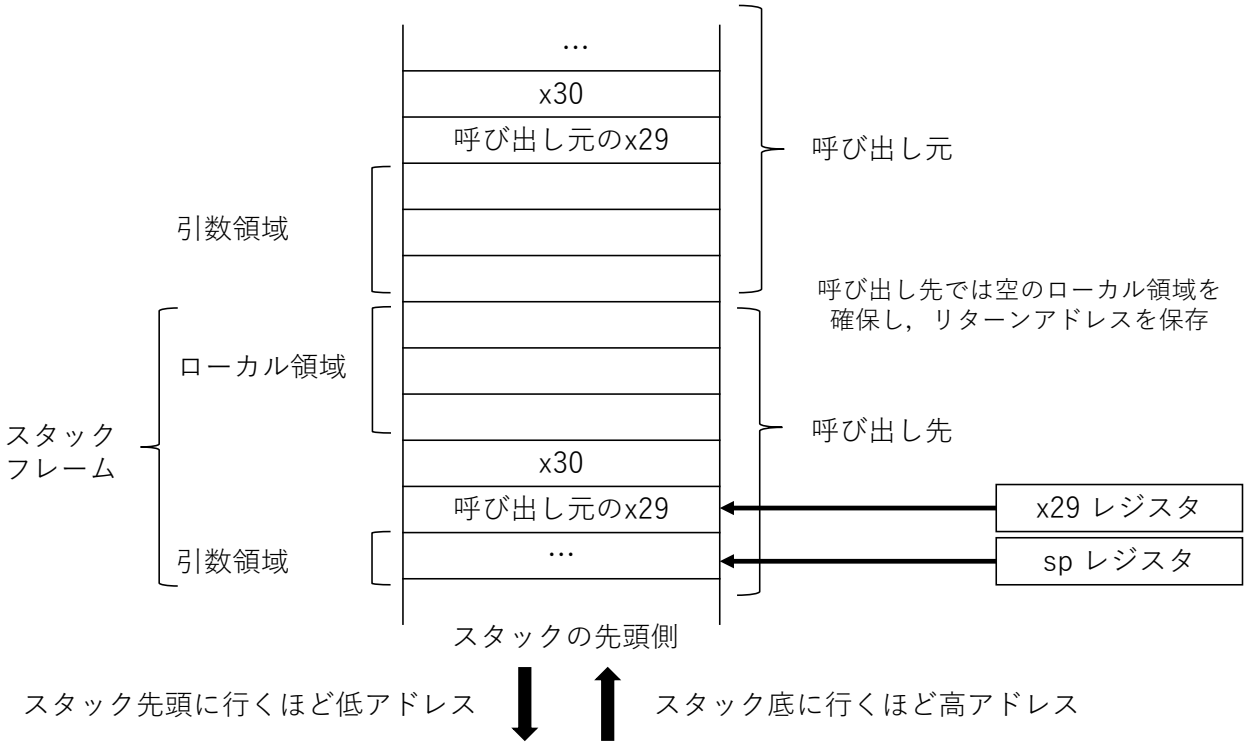

図 2.2: AArch64 におけるコールスタックの構造

2.2 スタックアンワインド

スタックアンワインドとは直訳するとスタックの巻き戻しという意味になる．すなわち，プログラムの実行中，実行後に，ある時点のスタックの状態に戻すことをスタックアンワインドという．そのためには，レジスタの値がいつの段階で何であるかを記憶しておく必要がある．この情報は，デバッガがスタックトレースを表示するということだけではなく，プログラムの解析ツールや高級プログラミング言語のランタイムに使われる．実際に C++では，この情報を使って例外処理機能を実装している．例外処理を例に上げて具体的にアンワインド情報（デバッグ情報）とはどんな情報であるかを説明する．

ソースコード 2.1: C++例外処理の例

```
1 #include<iostream>
2
3 void fn2(){
4     int a = 1;
5     //fn2 process
6     throw 0;
7     //fn2 process
```

```
 8 }
 9
10 void fn1(){
11     int a = 1;
12     //fn1 process
13     fn2();
14     //fn1 process
15 }
16
17 int main(){
18     //main process
19     try{
20         fn1();
21     }
22     catch(int e){
23         std::cout << "error" << e << std::endl;
24         //exceptions
25     }
26     //main process
27     return 0;
28 }
```

ソースコード 2.1 は C++言語における例外処理を使ったプログラムである．try ブロックの中に例外（エラー）が発生しうる処理を記述し，catch ブロックの中にその例外を処理する内容を書く．例外が発生したときには，エラーコードが catch ブロックの変数宣言した変数にキャッチされ，例外処理がなされる．ソースコード 2.1 では try ブロックで fn1 関数に実行が移り，fn1 関数から fn2 関数に実行が移っている．fn2 関数内で throw が実行されると，main 関数内の catch ブロックまで実行を戻す．この戻す流れでスタックアンワインドの処理が必要である．このとき，正しくアンワインドするためには，

- fn2 関数から main 関数に実行を戻すための正確なリターンアドレスの保存場所
- fn2 関数・fn1 関数で書き換えられた callee-saved レジスタの関数呼び出し前の値の保存場所

の情報が保存されていなければ，実行の制御とレジスタの復帰ができない．このアンワインド情報（デバッグ情報）は後述するバイナリファイル中の.eh_frame セクションに記載されている．

2.3 DWARF

2.3.1 ELF と DWARF

ELF はオブジェクトファイルや実行ファイルの形式である．これらのファイルは，プログラムのリンクとプログラムの実行に関与し，ELF では，これら二つのニーズに対応したファイルの内容を並べて表示される．一方，DWARF はソースレベルのデバッグをサポートするために使われるデバッグファイル形式である．DWARF は多くの言語の要件に対応しており，また，アーキテクチャに依存せずどのようなプロセッサや OS にも適応可能である．本論文では，DWARF に基づくデバッグ情報について扱うので，次節から詳しく説明する．

2.3.2 デバッグ情報

コンパイル時 -g オプションをつけたときに生成される DWARF 形式のデバッグ情報には以下のセクションがある．

.debug_info
DWARF 情報の核となるセクション．

.debug_line
行番号の情報をもつセクション．

.debug_loc
値の格納場所の情報をもつセクション．

.debug_str
.debug_info で使われる文字列テーブルを保存するセクション．

.debug_frame
コールフレームの情報をもつセクション．

本論文で扱うデバッグ情報は.eh_frame セクションである．この情報は.debug_info とほとんど同じ形をしており，実際にこの情報を使ってスタックアンワインドが実行される．DWARF 形式で記述されているが，バイナリファイルの ELF 形式で記述されたセクションの間に存在しており， -g [5] オプションをつけなくても生成される情報である．バイナリファイルに存在しているために readelf -w コマンドで情報を確認可能である．

2.3.3 .eh_frame

.eh_frame は，各 LOC において callee-saved レジスタに保存されている値がコールスタックのどのアドレスに保存されているかを示す情報である（AArch64 の場合は x29，x30 の情報も含む）．この情報を頼りに，C++言語などの高級言語のスタックアンワインドが実装されている．.eh_frame は Frame Description Entries (FDE) と呼ばれるセットの集合で構成されており，各 FDE はプログラムの関数それぞれに対応している．FDE ごとに異なる cananical frame address (cfa) と呼ばれるアドレスがあり，その FDE 内において変化することのない値である．x86_64 の場合，cfa は呼び出し元の call 命令が実行される直前の rsp の値になる．対して AArch64 の場合，cfa はその関数実行直前の sp の値になる．cfa はコールスタックのスタックポインタからのオフセットで表現されていて，callee-saved レジスタがコールスタックに保存されると，そのレジスタの値が保存されているコールスタック上のアドレスは cfa からのオフセットで表現される．つまり.eh_frame はレジスタの値が保存されたコールスタック上のアドレス，cfa，スタックポインタの相対的な位置関係を示す情報とみなせる．

表 2.3 は.eh_frame の情報を概念的に示したものである．一番左の列にはプログラム実行時にプログラムカウンタで保持されている値 LOC．その隣の列は cfa のアドレス．それ以降がそれぞれのレジスターのスタックアドレスの情報をもつ．各行は LOC 以外の列のいずれかが変化したときに生成をされる．表内の u は undefined を示す．例えば，LOC が 0x9d0 の行を注目すると，プログラムカウンタが 0x9d0 にセットされたとき，cfa の値はその時点での rsp からオフセット+16 だけ離れたアドレスで，rbp の値は cfa からオフセット-16 だけ離れたスタックアドレスに存在していることが読み取れる．また，プッシュされた callee-saved レジスタのデータがスタックから元のレジスタに復帰されるとき，表のアドレスデータは undefined に戻

表 2.3: x86_64 における.eh_frame の例

LOC	CFA	rbx	rbp	r12	r13	r14	r15
0x9cc	rsp+8	u	u	u	u	u	u
0x9d0	rsp+16	u	cfa-16	u	u	u	u
0x9ec	rsp+56	cfa-56	cfa-16	cfa-48	cfa-40	cfa-32	cfa-24
0xc90	rsp+8	u	u	u	u	u	u

る．つまり，LOC が 0xc90 の時点で，すべてのレジスタが関数呼び出し前の状態に復帰したことが読み取れる．

表 2.3 は，.eh_frame の理解のために表の形にしているが，実際には表を作成していくような DWARF 形式の命令と引数の集合で記述されている．この情報形式は，表 2.3 に比べて，更新されていないレジスタの情報が省略可能なので小さくて済む．例えば，図 2.3 の.eh_frame の元の情報はソースコード 2.2 のように記述される．

ソースコード 2.2: 表 2.3 の DWARF 記述

```
1  000000bc 000000000000003c 000000c0 FDE cie=00000000 pc=00000000000009cc
       ..0000000000000c94
2  DW_CFA_advance_loc: 4 to 00000000000009d0
3  DW_CFA_def_cfa_offset: 16
4  DW_CFA_offset: r6 (rbp) at cfa-16
5  DW_CFA_advance_loc: 24 to 00000000000009ec
6  DW_CFA_def_cfa_offset: 56
7  DW_CFA_offset: r15 (r15) at cfa-24
8  DW_CFA_offset: r14 (r14) at cfa-32
9  DW_CFA_offset: r13 (r13) at cfa-40
10 DW_CFA_offset: r12 (r12) at cfa-48
11 DW_CFA_offset: r3 (rbx) at cfa-56
12 DW_CFA_advance_loc1: 676 to 0000000000000c90
13 DW_CFA_restore: r3 (rbx)
14 DW_CFA_restore: r12 (r12)
15 DW_CFA_restore: r13 (r13)
16 DW_CFA_restore: r14 (rb14)
17 DW_CFA_restore: r15 (r15)
18 DW_CFA_restore: r6 (rbp)
19 DW_CFA_nop
```

以下，代表的な命令について説明する．厳密な定義は DWARF の公式ドキュメント [9] に記載のため，ここでは理解のために噛み砕いた説明をする．

DW_CFA_advance_loc

この命令の引数（to の前）の数を現在のコードアドレスに足したアドレス（to の後）の新しい行を作成する．cfa やレジスタの値は現在の行の情報が引き継がれる，行の作成が初めての場合は undedined となる．

DW_CFA_def_cfa_offset

現在の行の cfa 列のオフセットをこの命令の引数の値に書き換える．

DW_CFA_restore

AArch64 でよく見る命令であるが，この命令でスタックに保存された値が元のレジスタに復帰したことを示す．

DW_CFA_cfa_register

cfa を示すインデックスをスタックポインタから引数で指定されたレジスタに変更する．多くはフレームポインタが指定される，

DW_CFA_cfa

cfa を示すインデックスを引数で指定されたレジスタに変更する．多くは別のインデックスレジスタからスタックポインタに戻すときに使われる．

DW_CFA_nop

何もしない．

代表的な命令の説明に記載の通り，cfa のインデックスレジスタはスタックポインタからフレームポインタへの切り替え，またその逆が起こることがある．これはソースコード 2.3 のようにスタック上に動的に割り当てられる可変サイズのデータ構造が存在する場合に起こる．ソースコード 2.3，4 行目の i の値によって，for 文内の配列 z の要素数が変化している．それすなわち，使用するスタックの領域が変化するため，同じコードアドレスの命令を実行していてもスタックポインタの値が異なり，固定された cfa の値を表現することができない．そのため，同じく固定されたアドレスを指すフレームポインタが cfa インデックスとして切り替わるのである． -fomit-frame-pointer オプションによってフレームポインタが使用されていない場合でも，スタックの動的割り当ての構造が存在する関数内では，オプションに反してフレームポインタが使用される．

ソースコード 2.3: スタック上に動的に割り当てられる可変サイズのデータ構造の例（C 言語）

```
1  void main(){
2    int i;
3    int b_max = 10;
4    for ( i = 0; i < b_max; i++ ) {
5        int v[i] ;
6
7          ...
8
9    }
10 }
```

第3章　既存研究

3.1　Theseus

Theseus [3] は Kevin Boos らによって開発された Rust 言語の新しい OS である．Theseus における貢献は，以下の 2 つである．

1. 実行時の境界が明確に定義された多数のコンポーネントが，互いに状態を保持することなく相互作用する OS 構造である．

2. Rust 言語の機構を用いて OS そのものを実現し，コンパイラが OS のセマンティクスに関する不変性を強制可能．

Theseus はセルと呼ばれる小さな個別のコンポーネントからなるシステムアーキテクチャを規定しており，これは実行時に構成したり，交換したりすることが可能である．セルは実装時には Rust クレート，コンパイル時には一つのオブジェクトファイル，実行時にはセクションごとに境界とそのセクションのメタデータをもつメモリ領域の集合として存在している．注目すべきは，アプリケーションやライブラリなどのすべてのソフトウェアが OS のコアコンポーネントと同一のアドレス空間で共存し，単一の特権レベルで実行可能である．システムの安全性や独立性，整合性の確立は，既存の OS ではハードウェア保護と特権レベル，実行時のチェックに基づいたのに対して，Theseus は Rust の所有権モデルによって，メモリの安全性と管理を担い，セル同士の干渉を最小限にすることで実現している．

Rust 言語の機構を用いることで次の利点がある．

- コンパイラがリソース管理を引き受けることで，OS が維持しなければならない状態管理を減らし，各セルの独立性を強化可能．

- コンパイラがプログラムの挙動を把握し安全性をチェックすることで，すべてのソフトウェアで実行時ではなく，コンパイル時点でエラーを把握可能．

Theseus では DWARF 標準に基づくアンワインド方法で，既存のアンワインドライブラリを使わず，Rust で一からアンワインド機能を実装している，Theseus はアンワインド機能を次の目的で使用する．

アンワインド機能の目的

目的 1 リソース開放の保証

目的 2 フォールトリカバリ

目的 1 も目的 2 も，Rust 言語機構を利用して安全にリソースを使用している前提が崩れた場合を想定している．まず，目的 1 であるが，リソースの使用を制限しながら運用することだけでなく，参照されなくなったリソースを正しく開放することもメモリ安全につながる．リソースが開放されないというリソースリークを防ぐために，スタックアンワインド機能を使い，獲得したリソースが通常実行でも例外実行時で

も常に開放されるように実装している．ガベージコレクタとは違い，通常の実行性能に影響を及ぼすことなくリソース開放を行える．次に，目的 2 であるが，言語レベルの例外や CPU などのハードウェア起因の障害時に，復旧作業の最初の段階でアンワインド機能が使われる．問題のあるタスクをアンワインドして，リソースを完全に開放，タスク実行前の状態まで回復することでこれに対処している．このように，問題のあるタスクを巻き戻すことで，そのタスクで使用されていたリソースを共有する別のタスクとの隔離を実現し，別のタスクの実行を継続可能である．

TheseusOS の研究では，フォールトリカバリについて性能評価をしており，80 万個のフォールトをプログラムに注入すると，そのうち 665 件の観測可能な障害が発生し，この内，フォールトリカバリが回復したのは 461 件，失敗したのは 204 件であった．この 204 件のうち，62 件はアンワインド機能自体に障害が発生したという評価であった．

3.2 検査アルゴリズム

x86_64 アーキテクチャ用の.eh_frame 検査ツールの設計・開発をおこなった既存研究 [2] について説明をする．このツールは，バイナリファイルと.eh_frame を動的にクロスチェックし，.eh_frame の検出困難なバグを特定可能である．システムの説明の前に，バイナリファイルを逆アセンブルしたコードと.eh_frame がどのような関係にあるか説明する．図 3.2 は，とある x86_64 アーキテクチャプログラムのアセンブリコードとそれに対応した.eh_frame の cfa，rbp 列とリターンアドレスの情報を並べている．

0	<fn1>:		CFA	rbp	ra
1	push	%r15	rsp+8	u	cfa-8
2	push	%r14	rsp+16	u	cfa-8
3	mov	$0x3, %eax	rsp+24	u	cfa-8
4	push	%rbx	rsp+24	u	cfa-8
5	push	%rbp	rsp+32	cfa-40	cfa-8
6	sub	$0x68, %edi	rsp+40	cfa-40	cfa-8
7	cmp	$0x1, %edi	rsp+144	cfa-40	cfa-8
8	movl	$0x0, 0x14 (%rsp)	rsp+144	cfa-40	cfa-8
9	je	.L2	rsp+144	cfa-40	cfa-8
10	add	$0x68, %rsp	rsp+144	cfa-40	cfa-8
11	pop	%rbp	rsp+40	cfa-40	cfa-8
12	pop	%rbx	rsp+32	u	cfa-8

図 3.1: アセンブリコードと対応する.eh_frame 情報

一行目，.eh_frame の CFA 列，リターンアドレス（ra）の情報であるが，2.3.3 節で説明の通り，x86_64 アーキテクチャの cfa は call 命令によってリターンアドレスがスタックに積まれる直前の rsp の値になる．すなわち，関数呼び出しされた直後，rsp は cfa からリターンアドレス分 8 バイトだけずれるので，cfa は rsp を用いると rsp+8 と表される．このとき，リターンアドレスは rsp が指すアドレスに積まれているので，cfa を用いると cfa−8 と表されることになる．また，リターンアドレスはその関数が終了するまで固定されたアドレスに位置し続け，cfa もその関数実行中は固定されているため，常に cfa−8 のアドレスにリターンアドレスは位置し続ける．これは，基本的に callee-save レジスタも同様である．図 3.2 の 5 行目で，rbp が

コールスタックにプッシュされて.eh_frame にもスタックアドレス cfa−40 が反映されているが，そのスタックアドレスは変動することがないのでスタックからポップされるまで同じアドレスが記録される．

.eh_frame 情報において最も重要なことは，rsp の値を変化を追跡することである．callee-saved レジスタやリターンアドレスは cfa からのオフセットで表されるが，cfa は rsp からのオフセットで表現される，つまり，rsp の値の変化を追跡できなくなったとき，すべてのアドレス情報が誤りになってしまう．この rsp の値の変化は CFA 列に反映される．例えば，図 3.2 の 1 行目は，プッシュ命令なので rsp の値が −8 される．cfa は固定された値であるので，2 行目の CFA 列で rsp の変化を打ち消すようにオフセットが +8 されている．

```
0  <fn1>:                    CFA       ra
1   push   %r15              rsp+8     cfa-8
2    …                       rsp+16    cfa-8
3    …                       …
4   pop    %rbx               rsp+16    cfa-8
5   retq                      rsp+16    cfa-8
```

図 3.2: 誤った.eh_frame 情報

図 3.2 は，この rsp の更新が誤った.eh_frame の例を示す．4 行目，rbx の値がポップされているのにも関わらず，CFA 列は更新されていない．5 行目の retq 命令で rsp が指すリターンアドレスの値をポップするはずなのだが，この.eh_frame の情報では，リターンアドレスは (rsp+16) − 8 =rsp+8 に存在することになっているため，この.eh_frame は正しいリターンアドレスのスタックアドレスを記録できていない．検査アルゴリズムでは，このような.eh_frame の誤りを検出する．

Algorithm 1 検査アルゴリズムの擬似コード

```
1: function MAIN
2:     abstract_state = {}
3:     eh_frame_table = perse_eh_table(file)
4:     gdb_advance_to_main(file)
5:     while True do
6:         (ip, instr) = gdb_get_current_instruction()
7:         regs = gdb_get_registers()
8:         eh_entry = get_eh(eh_frame_table, ip)
9:         if abstract_state.top() != compute_ra(eh_entry, regs) then
10:            error('Inconsistent table at'+ip)
11:        end if
12:        if instr == 'call' then
13:            abstract_state.push(regs['rsp']−8)
14:        else if instr == 'ret' then
15:            abstract_state.pop()
16:            if empty(abstract_state) then
17:                exit('Validation succesful')
18:            end if
19:        end if
20:        gdb_next_step
21:    end while
22: end function
```

Algorithm.1 は.eh_frame 検査アルゴリズムの擬似コードである．このシステムは，pyelftools [7] ライブラリを用いて.eh_frame 情報を読み込み，gdb デバッガを用いて検査対象プログラムを 1 命令ずつ実行する．abstract_state はスタックであり，call 命令が実行されるときには，引数となるリターンアドレスが保存されるスタックアドレス rsp−8 をプッシュし，ret 命令が実行されるときには，このスタックの先頭の値をポップする．このとき，abstract_state の先頭の値は，現在実行している関数のリターンアドレスが保存されているスタックアドレスになる．1 命令実行ごとに，abstract_state の先頭の実際のリターンアドレスと現在の rsp の値，読み込んだ.eh_frame によって計算されるリターンアドレスが等しくなるか確認することにより，.eh_frame の誤りが検出可能である．仮に，実際のリターンアドレスが保存されているスタックアドレスと計算したスタックアドレスが等しくない場合，前述の通り，rsp の値が変化する命令にも関わらず，.eh_frame 上の cfa 列のオフセットを書き換えていないという誤りが発生しているということになる．

3.3 合成アルゴリズム

x86_64 アーキテクチャ用の.eh_frame 合成ツールの設計・開発をおこなった既存研究について説明をする．この合成は，アセンブリプログラムテキスト情報のみで行うことが可能である．

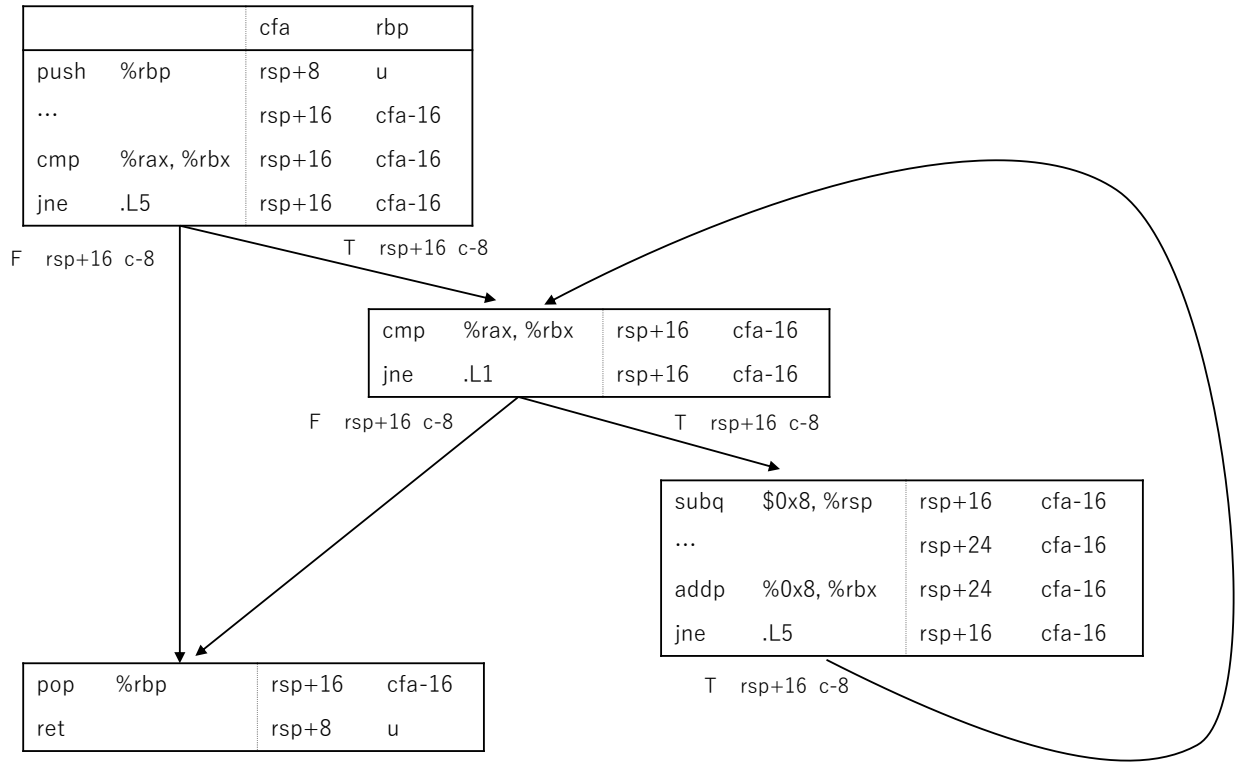

図 3.3: アセンブリコードと合成された.eh_frame の制御フローグラフ

まず，コンパイラが任意の命令でスタックサイズを静的に計算可能な関数，つまり，cfa が常に rsp からのオフセットで表される関数について考える．ここでは，callee-saved レジスタの情報は rbp のみを考える．まず，その関数のアセンブリコードから図 3.3 のように制御フローグラフを構築する．最初の基本ブロックの先頭は，検査アルゴリズムでも説明の通り，cfa は rsp+8 である．ここからアセンブリ命令を一行ずつ読み込んでいく．rsp の情報が変化する命令（push や subq $0x8, %rsp など）を読み込んだときは，cfa のオフセットを更新する．どれだけ値が変化するかは命令による．callee-saved レジスタは未定義の状態から，コールスタックにプッシュされる命令 (push %rbp など) を読み込んだとき，レジスタ列をスタックに保存されるアドレスに更新する．基本的には rsp の指すアドレスに保存されるので，cfa 列 rsp からのオフセットの負の数が cfa からのオフセットとなる．このようにして，最初の基本ブロックの.eh_frame が生成できたら，ブロックの最後の行を制御フローの矢印の指すブロックのはじめの行に伝搬させる．伝搬されたブロックにも同様の作業を行い，制御フローグラフでつながっているすべてのブロックについて.eh_frame を作成する．図 3.3 の最後のブロックのように，定義された callee-saved レジスタがスタックに保存した元の値に復帰する命令（pop）を読み込むと，そのレジスタ列を未定義に戻す．ちなみに制御フローグラフにしたがって情報を伝搬する際，ループ構造で見られるように，以前に伝搬したブロックに戻ることがある．このとき伝搬された行が以前に伝搬された行と衝突することはない．これは，コンパイラが任意の瞬間で，rsp から静的に計算されたオフセットでスタックに割り当てられた変数を参照可能であるという仮定に基づいている．そのため，このアセンブリコードは元より，制御フローグラフのマージポイントで各列のオフセットが不変になるように作られている．

制御フローグラフの作成やアセンブリ命令の読み込みには Binary Analysis Platform（BAP） [6] というツールが使われている．アセンブリ命令をそのまま解釈することは難しいが，BAP の提供するツールを用いれば，バイナリをアセンブリ命令よりも簡易な RISC ライク命令言語に分解することができ，レジスタやスタックメモリの書き換え一つにつき，一つの命令で表される．例えば，push　%rbp という命令は，rsp

の書き換えとコールスタックスタックの書き換えが同時に行われる命令だが，BAP ではこれらがそれぞれ単独の命令に分けられる．

次に，スタック上に動的に割り当てられる可変サイズのデータ構造が存在する関数，つまり，cfa が時に rbp からのオフセットで表される関数について考える．このときの問題点は，スタック上に動的に割り当てられる可変サイズのデータ構造が存在する関数か否かをまず判断しなければならないということである．その判断をアセンブリコードから行うのは難しい．そのため，この論文で提案されたアルゴリズムでは rsp-index モード，rbp-index モードという 2 パスアプローチを採用しており，rsp-mode での合成が失敗したときに rbp-mode へ移行する．rsp-mode とは，先程説明した，cfa が常に rsp からのオフセットで表されることを前提として.eh_frame を合成する方式である．この方式が失敗したとき，それはスタック上に動的に割り当てられる可変サイズのデータ構造が存在していることを意味しており，“ 必ず ”cfa が rbp からのオフセットで表されるタイミングが存在する．これを前提として，.eh_frame を合成する方式を rbp-mode とする．この 2 パスアプローチによって，合成対象の関数にスタック上に動的に割り当てられる可変サイズのデータ構造が存在するか否かを判断することは必要なくなる．rsp-mode が失敗したという判断は，rsp レジスタの更新に着目をする．rsp の更新は，通常 BAP 内部表現で rsp <- rsp + offset という形で表され，この形式における offset を CFA 列の更新に用いる．rsp 更新がこの形式ではない場合（ rsp <- rbp など）、rsp-index モードは失敗したと見なし，rbp-index モードに移行する．rbp-index モードでは，cfa のインデックスが rsp から rbp に変更される瞬間，またその逆を検知しなければならない．rsp から rbp への切り替わりは，rsp の値を rbp にコピーしたとき，BAP 内部表現で rbp <- rsp の命令を読み込んだときに行う．rbp から rsp への切り替わりは，rbp の値が書き換わるとき，BAP 内部表現で rbp <- ○○ を読み込んだ時に行う．この切り替わりにおける rsp からのオフセットはコードからは読み取れないが，実験的にこの切り替わりは関数の一番最後でしか行われないため，cfa = rsp + 8 と書き換える．

最後に，制御フローグラフのマージポイントにおける衝突について説明する．スタック上に動的に割り当てられる可変サイズのデータ構造が存在する場合，マージポイントでは衝突が生じる場合がある．これは，rbp が一方で未定義，もう一方では定義されている行をマージしようとする場合である．マージポイントは等しい行でマージされなければならないため，マージされた行の rbp は未定義として統一することで，整合関係を安全に弱める事が可能である．しかし，ここで情報の欠損が生じてしまうのだが，実験では，clang でコンパイルされたプログラムの稀なケースで，最終ブロックでのみ，このようなマージを発見している．そのため，最終ブロックでマージされる行でのみ，欠損の生じるマージを許すことにする．

Algorithm 2 合成アルゴリズムの擬似コード

```
1: function MAIN
2:     eh_frame = {}; visited_blks = {}
3:     last_row = cfa = rsp + 8; rbp = u;
4:     for all function f in the binary do
5:         entry_blk = {}, cfg = BAP_build_cfg(f)
6:         success = DFS(entry_blk, cfg, last_row, eh_frame, 'rsp-mode')
7:         if not success then
8:             eh_frame = ; visited_blks = {}
9:             success = DFS(entry_blk, cfg, last_row, eh_frame, 'rbp-mode')
10:            if not success then
11:                error('synthesis failed')
12:            end if
13:        end if
14:    end for
15:    return eh_frame
16: end function
17:
18: function DFS(blk, cfg, last_row, eh_frame, mode)
19:    if not consistent(last_row, eh_frame,(first_ip(blk))) then
20:        error('inconsistency at' + fisrt_ip(blk))
21:    end if
22:    if not in(blk, visited_blks) then
23:        for all ip in blks do
24:            instr = instruction_at(ip, blk)
25:            eh_frame(ip) = last_row
26:            success, last_row = synthesize_row(ip, instr, last_row, mode)
27:            if not success then
28:                return false
29:            end if
30:        end for
31:        add(blk, visited_blk)
32:        for all s_blk in successors(blk, cfg) do
33:            success = DFS(s_blk, cfg, last_row, eh_frame, mode)
34:            if not success then
35:                return false
36:            end if
37:        end for
38:    end if
39:    return true
40: end function
```

第4章　AArch64上での設計と実装

本研究の環境は，以下の通りである．

表 4.1: 実行環境

CPU	ブロードコム BCM2711, Quad-core Cortex-A72 (ARM v8) 64-ビット SoC @1.5GHz
メモリ	4GB
OS	Ubuntu 21.04
プログラム	Python 3.9.5

4.1　検査アルゴリズム

4.1.1　設計

AArch64 アーキテクチャにおいても，3.2 節で提案されたように，検査対象バイナリファイルを一命令ずつ実行し，その都度.eh_frame と比較する方式を採用している．.eh_frame の構成や検査方法の大枠は 2 章，3 章を参照されたい．本節では，我々が提案するシステムと [2] の異なる点に絞って説明する．

x86_64 アーキテクチャ用既存アルゴリズムの適用における問題点

命令セットの違いは当然のこととして，Algorithm.1 が AArch64 アーキテクチャにそのまま適応できない根拠は次の二点である．

問題点

問題点 1 呼び出し先関数でリターンアドレスが保存されるスタックアドレスは呼び出し元からは不明であること．

問題点 2 .eh_frame を検査できない瞬間が発生すること．

まず，問題点 1 について説明する．図 4.1.1 は，x86_64 の関数呼び出しコードとそれに対応するスタック，AArch64 の関数呼び出しコードとそれに対応するスタックを表している．

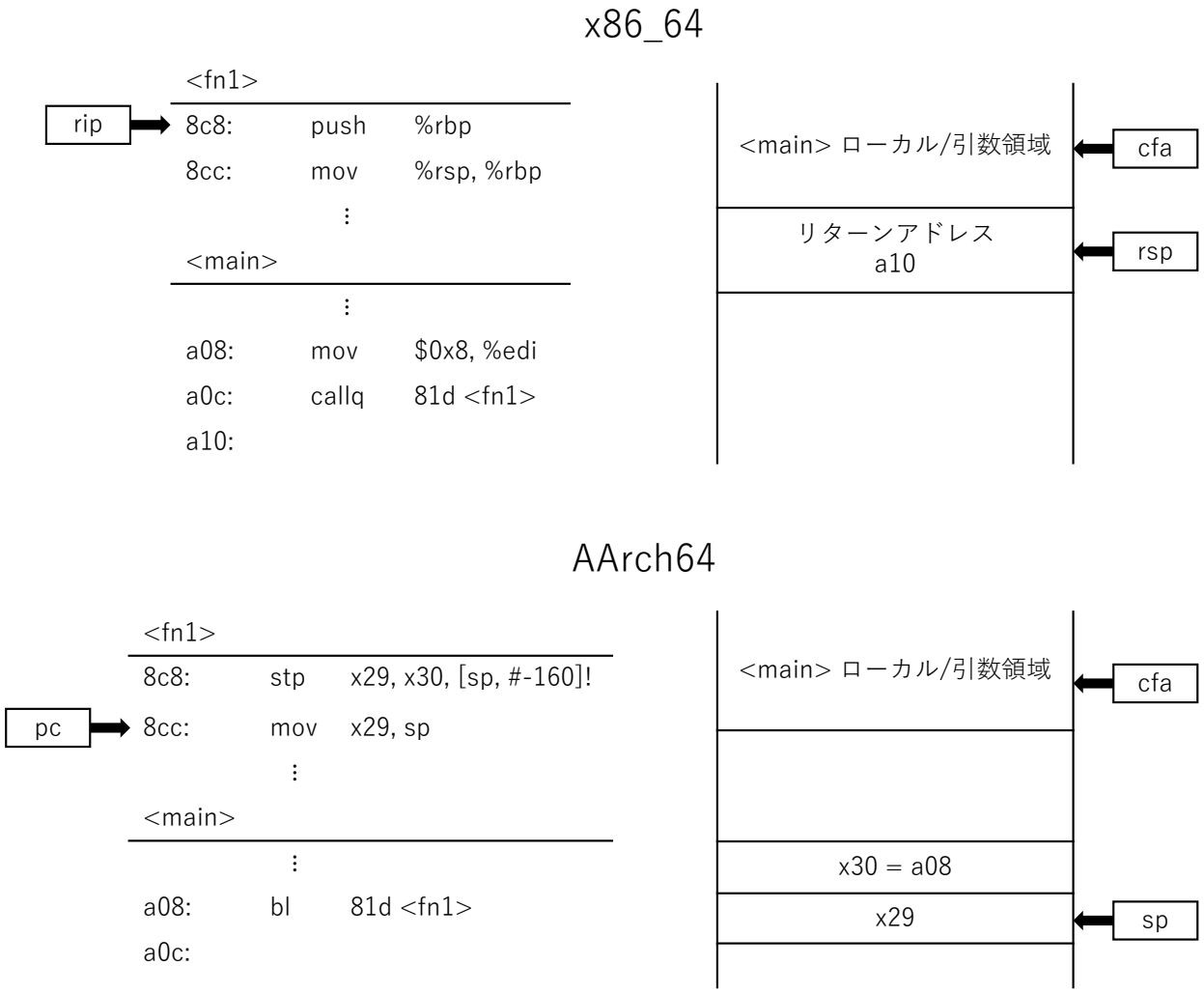

図 4.1: x86_64 と AArch64 の呼び出し規則の違い

x86_64 の場合，call 命令実行後リターンアドレスがスタックの先頭にプッシュされるため，関数呼び出し直前の main 関数側でリターンアドレスが保存されるスタックアドレスは cfa−8 とわかっている．これにより，Algorithm.1 において，call 命令を実行する前から abstract_state にスタックアドレスを保存することが可能である．対する AArch64 では，bl（または blr）命令が実行されると x30 レジスタにリターンアドレスが一度保存され，この x30 レジスタの値をスタックに保存する責任を負うのは関数呼び出し後の fn1 関数側である．また，AArch64 ではコンパイラが事前に fn1 関数で必要となるローカル領域，引数領域の大きさを計算し，スタックポインタの更新，x30 レジスタの値の保存が行われる．つまり，関数呼び出し直前の main 関数側では，x30 レジスタの値がスタックのどこに保存されるか事前に知ることができないのである．

次に，問題点 2 について説明する．関数実行初めの x30 レジスタの値をスタックに保存するまでの間や，関数呼び出し命令が存在しない関数ではコールスタックに x30 レジスタの値を保存していない．そもそも 3.2 節では，スタックポインタが更新されないことで.eh_frame のスタックアドレス情報がずれてしまうというバグを，プログラム実行時 “常に”，“自明なスタックアドレスに” 保存されているリターンアドレスをたどることによって検出している．しかし，このリターンアドレスが存在しない瞬間が存在する AArch64 ではスタックアドレス更新のバグを検出できないタイミングが存在してしまう．

AArch64 アーキテクチャに適したアルゴリズム

我々が提案する AArch64 用の検査システムは，[2] を拡張し，より広いバグを検出可能なシステムである．具体的には，次の 3 点の機能を追加した．

機能

機能 1 .eh_frame における callee-saved レジスタのスタック保存情報通りに，実際の検査対象プログラムが実行されていることを検査する機能．

機能 2 .eh_frame に記載されていない callee-saved レジスタのスタック保存が，実際の検査対象プログラムで実行されていないことを検査する機能．

機能 3 スタック保存された callee-saved レジスタの値が関数終了時にレジスタ復帰していることを検査する機能．

機能 1，機能 2 は一見同じ機能のように思えるが，検査するバグの種類が異なる．機能 1 は，.eh_frame に記載されている callee-saved レジスタが保存されているスタックアドレス自体が間違っているバグや，実際にはレジスタの保存がされない実行であるにも関わらず，.eh_frame にはそのレジスタがスタックに保存されているという誤記のバグを検出する．機能 2 は，実際の実行時に str，stp 命令によってレジスタの値がスタックに保存されているが.eh_frame に反映されていないというバグを検出する．.eh_frame 上でスタックアドレスが記載されているレジスタは機能 1 の検査を，記載されていないレジスタは機能 2 の検査を実施することで，すべてのレジスタにおいて，実際の実行状況と.eh_frame の情報が正確に対応しているか検査することが可能である．

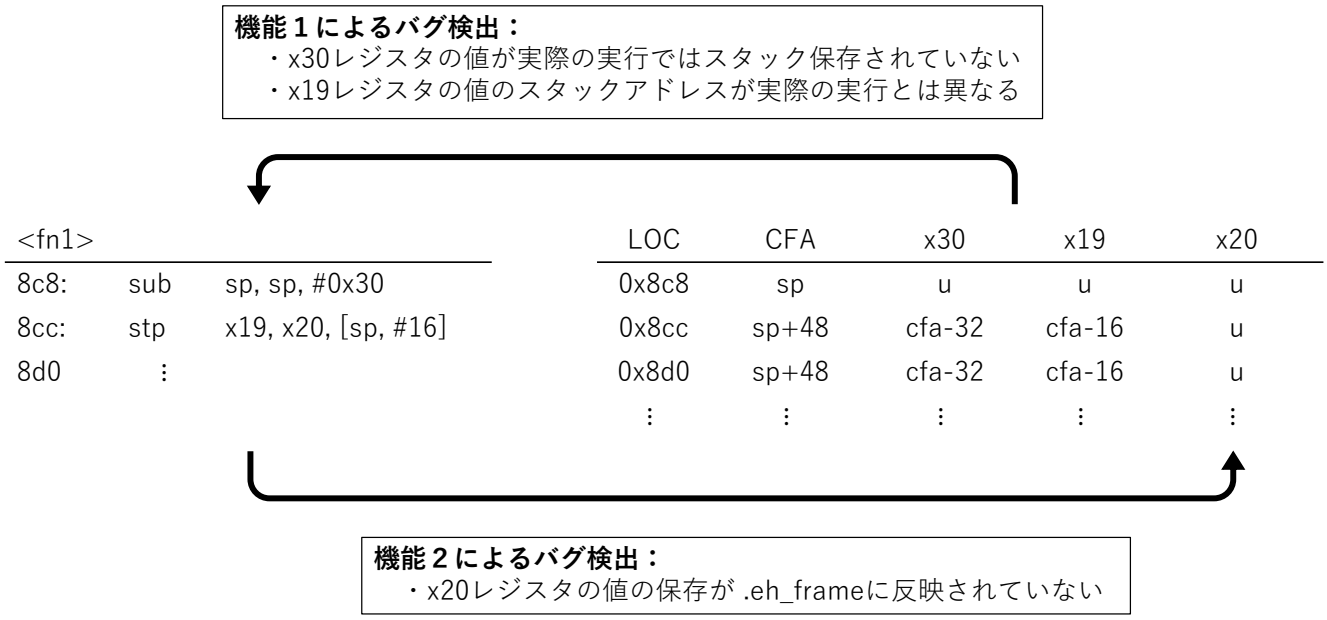

図 4.2: 機能１と機能２の例（実際の実行（左），対応する.eh_frame（右））

なお，[2] で検出されるスタックポインタ更新のバグは機能 1 に含まれており，callee-saved レジスタが保存されているスタックアドレス自体が間違っているというバグの一つの要因となる．この検出はリターンアドレスに依存しておらず，他の callee-saved レジスタに責任が分散される．仮に，レジスタ情報が全くない関数の.eh_frame FDE はスタックアンワインドに利用しないためスタックポインタ更新のバグ検出は必要なく，レジスタ情報が初めにはなく後から追加される関数の.eh_frame FDE はレジスタ情報を追加したすぐ次の検査でバグを検出可能ため，問題点 2 を克服している．

Algorithm 3 検査アルゴリズムの擬似コード

```
1: function MAIN
2:     abstract_state = {}
3:     error_regs = {}
4:     eh_frame_table = perse_eh_table(file)
5:     gdb_advance_to_main(file)
6:     initial_process(abstract_state, eh_frame_table)
7:     while True do
8:         (ip, instr) = gdb_get_current_instruction()
9:         regs = gdb_get_registers()
10:        eh_entry = get_eh(eh_frame_table, ip)
11:        for all reg in regs do
12:            if abstract_state.top() != compute_regs(eh_entry, reg) then
13:                error('Inconsistent table at'+ip)
14:            else if previous_instr == 'str' or 'stp' reg then
15:                error_regs.push(reg)
16:            else if reg in eh_entry then
17:                error_regs.pop(reg)
18:            end if
19:        end for
20:        if instr == 'bl' or 'blr' then
21:            if any(error_regs) then error('Lack of information at' + error_regs)
22:            end if
23:            gdb_next_step
24:            abstract_state.push(all regs)
25:        else if instr == 'ret' then
26:            if any(error_regs) then
27:                error('Lack of information at' + error_regs)
28:            end if
29:            current_state = abstract_state.pop()
30:            if current_state == loaded(eh_entry, regs) then
31:                if empty(abstract_state) then
32:                    exit('Validation succesful')
33:                end if
34:            else
35:                error('Lack of information at'+ip)
36:            end if
37:        end if
38:        previous_ip, previous_instr = ip, instr
39:        gdb_next_step
40:    end while
41: end function
```

Algorithm.3 は，検査アルゴリズムの擬似コードである．特筆すべきは 20 行目の関数呼び出し時の処理

である．3.2 節では，リターンアドレスが保存されているスタックアドレスを abstract_state に保存していたが，Algorithm.3 では，すべての該当するレジスタ，かつ “レジスタの値そのもの” を abstract_state に保存している．レジスタの値そのものの情報によって.eh_frame を検証可能とめ，問題点 1 を克服した．

11 行目の for 文内では該当するすべてのレジスタ（x30，x29 ，x19 x28）に対して，12 行目で機能 1，14 行目で機能 2 の検査を行っている．12 行目の compute_regs 関数では.eh_frame のレジスタが保存されているスタックアドレスを読み込み，コールスタック上のそのアドレスに保存されている値を返す．それを abstract_state に保存されたレジスタの値と比較して検査を行っている．

また，14 行目の検査で引っかかったとしても，そのエラー情報を error_regs というリストに保存しておき，bl，blr，ret 命令を実行する直前のタイミングで出力する．この理由は，実際の実行で callee-saved レジスタの値のスタック保存がされたとき，その直後の LOC の値で.eh_frame に反映されるわけではなく，それよりいくらか後の LOC 値で反映されることがあるためである．error_regs に保存されたレジスタが.eh_frame に反映された際には，16，17 行目の処理で破棄され，関数呼び出しまたは関数終了までにこのエラー情報が残っていたときには，スタック保存の情報が.eh_frame に反映されなかったとしてエラー情報を出力する．

さらに 25 行目の関数終了時の処理では，機能 3 の検査を行っている．単に abstract_state の先頭をポップするだけではなく，.eh_frame でレジスタ復帰したとされるレジスタが，ポップした情報通りに復帰しているかを検査をしている．この検査でバグを検出をすると，.eh_frame 上ではスタックからレジスタに値が復帰しているにも関わらず，実際の実行ではそれが行われていない，または，.eh_frame に記述されたスタックアドレスとは異なるアドレスの値をレジスタ復帰されていることを意味する．

なお，本システムは main 関数に該当する.eh_frame FDE も検査するが，gdb によって検査対象プログラムを実行（start コマンド）すると，メイン関数の x30，x29，callee-saved レジスタが保存され終わった状態でブレークされるため，スタックに保存される直前のレジスタの値を保存することができない．そのため本システムでは，メイン関数に該当する FDE 上のはじめにレジスタの値が保存されるアドレス情報が正しいという前提で，そのアドレスに保存されている値を 6 行目 initial_process の処理で abstrct_state にプッシュし，メイン関数の検査に用いている．そのアドレス情報が誤っている場合や不足している場合は，メイン関数終了時，30 行目の検査に引っかかりエラーを出力する．

4.1.2 実装

本システムは Python で実装し，3.2 節と同様に GNU デバッガ (gdb) [4] と pyelftools [7] を活用している．

gdb は文字通りデバッガで様々なプログラム言語に対応していることから採用した．本システムでは監査対象プログラムファイルを読み込み，メイン関数実行開始，1 ステップ実行，レジスタ情報や逆アセンブルした命令の抽出を行っている．

ソースコード 4.1: Python スクリプトファイルから gdb を操作する例

```
1 import gdb
2
3 gdb.execute('file testfile') # 検査対象プログラムの読み込み
4 gdb.execute('start') # メイン関数実行開始
5 gdb.execute('stepi') # 1 ステップ実行
6 instr = gdb.execute('x/1i $pc',to_string = True) # 逆アセンブルした命令の抽出
7 regs = gdb.execute('i r', to_string=True) # レジスタ情報の抽出
```

pyelftools は ELF 形式ファイルや DWARF 形式デバッグ情報分析のための様々なツールを提供する Python ライブラリである．本システムでは，.eh_frame セクションの読み込みに使用し，ソースコード 4.2 のよう

に書かれている．

ソースコード 4.2: pyelftools による.eh_frame の読み込み

```
1  from elftools.elf.elffile import ELFFile
2
3  def process_file(filename):
4      print('Processing file:', filename)
5      with open(filename, 'rb') as f:
6          elffile = ELFFile(f)
7
8          if not elffile.has_dwarf_info():
9              print(' file has no DWARF info')
10             return
11
12         dwarfinfo = elffile.get_dwarf_info()
13         fdes = dwarfinfo.EH_CFI_entries()
14     return fdes
15
16 eh_frame = process_file("testfile")
```

また，実際に検査対象プログラム実行中，プログラムカウンタの値は実行ファイルで示される値と一定値の差がある．これは Linux カーネルの Position-independent executable（PIE）によるもので，PIE 機能は実行ファイルに対してランダムなベースアドレスを規定している．我々の実行環境においてベースアドレスは 0xaaaaaaaa0000 であることが判明したため，プログラムカウンタの値からこのベースレジスタ分引いた値を本来のプログラムカウンタの値として使用している．

4.2 合成アルゴリズム

AArch64 アーキテクチャにおいても，3.3 節の考え方やアルゴリズムを踏襲してシステム開発を行った．本節では，アルゴリズム.2 の 26 行目，synthesis_row でのアンワインド行生成処理における CPU アーキテクチャの呼び出し規則の違いが出る点に絞って説明をする．また，3.3 節で利用している BainaryAnalysisPlatform（BAP） [6] を本システムでも利用としたところ，ツールの不具合が見つかった．その詳しい内容と回避方法について 4.2.2 節にまとめる．

4.2.1 設計

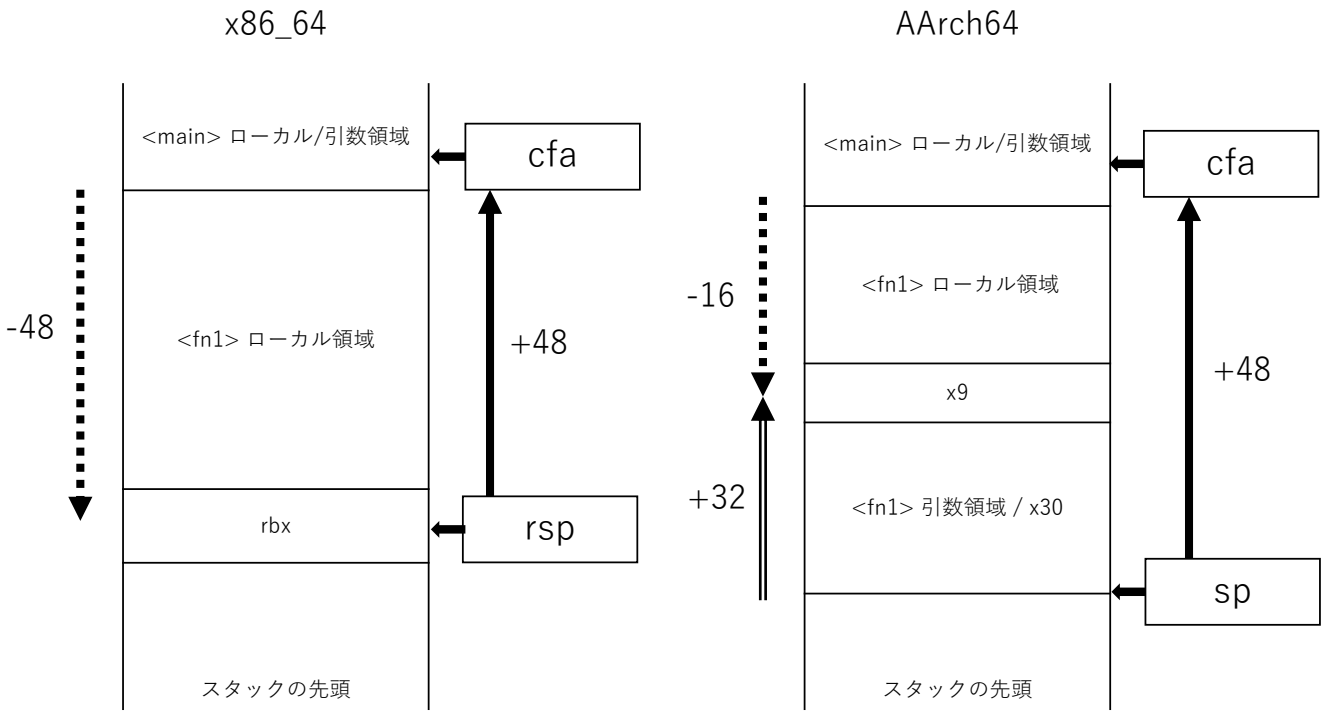

図 4.3: x86_64 と AArch64 のスタック保存の違い

図 4.2.1 は x86_64，AArch64 アーキテクチャで，callee-saved レジスタを保存したときのスタック状態を表している．x86_64 の図は，push %rbx 命令を実行した直後の状態である．x86_64 の場合，rsp がスタックの先頭を必ず指していて，それは rbx レジスタの値が保存されたスタックアドレスになっている．そのため，cfa の rsp からのオフセットが +48 ならば，rbx レジスタの値が保存されたスタックアドレスの cfa からのオフセットは −48 となる．つまり，.eh_frame レジスタ列を undefined から定義する際には，その時点での.eh_frameCFA 列のオフセットの符号を逆転させた値をオフセットに指定することで各レジスタのアンワインド情報が生成可能である．対する AArch64 では，push 命令の代わりに str 命令や stp 命令が使用される．検査アルゴリズムの 4.1.1 節で説明の通り，関数呼び出し後の sp はその関数が使用する領域分移動し，str，stp 命令でその sp をベースレジスタとしてどれだけ離れたアドレスに保存をするのかという表現でスタックに値が保存される．AArch64 の図は，str x9, [sp, 32] 命令を実行した直後のスタック状態である．この命令は，sp+32 で計算されるアドレスに x9 レジスタの値を保存するという命令なので，cfa の sp からのオフセットが +48 ならば，x9 レジスタの値が保存されたスタックアドレスの cfa からのオフセットは $32 - 48 = -16$ となる．このように，AArch64 の場合は str，str，stp の中間表現命令の sp からのオフセットに該当する値を読み込んで，そこから CFA 列のオフセットを引くことでレジスタ列のオフセットを求めている．

3.3 節で rsp-index モード，rbp-index モードについて説明しているが，本章では AArch64 のレジスタ名を使って，それぞれ sp-index モード，x29-index モードと呼ぶことにする．AArch64 アーキテクチャも x86_64 アーキテクチャと同様に，フレームポインタ x29 にスタックポインタ sp の値をコピーする（mov x29, sp）とき，.eh_frame CFA 列のベースレジスタが sp から x29 に切り替わることを確認している．この命令は，関

数初めにスタックフレームのフレームアドレスを x29 に設定する処理に見られる．しかし，これは sp が呼び出し元のフレームアドレスを保存するスタックアドレスを指していることが前提となっており，AArch64 アーキテクチャでは常にそれが成り立っているとは限らず，代わりに sp+offset の形でフレームアドレスを指定して x29 に保存する（add x29, sp, offset）ことがある．そのため，本システムでは，x29-index モードでこの命令もベースレジスタ切り替えを検知するように設計している．

逆に.eh_frame CFA 列のベースレジスタが x29 から sp に戻る切り替えは，x86_64 アーキテクチャと同様にフレームポインタ x29 の値が書き換わるときであることを AArch64 アーキテクチャでも確認している．この x29 の書き換えは，関数のリターン直前，呼び出し先関数のフレームアドレスから呼び出し元関数のフレームアドレスに x29 が書き換えられるときに見られる．このときの sp の値は，CFA 列のベースレジスタが sp から x29 に切り替わるときの sp と等しくなるため，この切替直前の CFA 列のオフセットを，.eh_frame CFA 列のベースレジスタが x29 から sp に戻ったときのオフセットに使用する設計となっている．

しかしながら，CFA 列のベースレジスタが x29 から sp に戻る切り替えが x29 の値が書き換わるときではない場合を発見している．そのプログラムは，関数初めに add x29, sp, offset 命令で CFA 列ベースレジスタが sp から x29 に切り替わり，関数終了時の calle-saved レジスタを復帰させる直前に sub sp, x29, offset 命令で CFA 列ベースレジスタが x29 から sp に切り替わっていた．十分なテストケースがないため，本システムではこのケースのベースレジスタ切り替わりの検出を実装できていないが，x29 の書き換えタイミングでの切り替わりで対応している．切り替わりタイミングがずれていても，合成対象プログラムのアセンブリコードと合成したアンワインド情報の整合性が崩れることはない．

4.2.2 実装

本システムは， [2] でも使用されている Bainary Analysis Platform（BAP） [6] を使用している．BAP は，x86, x86_64, ARM, MIPS, PowerPC 環境でバイナリファイルに対して様々なツールを提供している．本システムにおいては，対象プログラムの制御フローグラフの作成，中間表現命令変換に利用している．

BAP で実行ファイルを解析した結果は，まず関数別に情報が分けられ，各関数の中では基本ブロック別に分けられている．基本ブロック内では，実行コードを中間表現変換された命令セットと，そのブロックと制御フローがつながる次の基本ブロックの識別アドレス（基本ブロック先頭の中間表現命令 LOC）の情報が含まれている．中間表現変換された命令セットはアルゴリズム 2，24 行目 for 文でアンワインド情報作成に，次の基本ブロックの識別アドレスは 32 行目からの再帰的にすべての基本ブロックを走査する処理に使用されている．

中間表現の実行命令情報は，中間表現命令の LOC，その命令に該当する逆アセンブリコード，書き換え対象レジスタ/メモリ，書き換える値の 4 つの情報に分かれている．

ソースコード 4.3: 中間表現命令の例

```
1  """
2  c90: stp x29, x30, [sp, #-128]!を中間表現に変換したコード↓
3
4  """
5
6  """
7  Def(
8    # 中間表現命令ＬＯＣ
9    # 中間表現命令に該当する逆アセンブリコード情報
10   # 書き換え対象レジスタ／メモリ
11   # 書き換える値
```

```
12 )
13 """
14
15 # x29("FP") のスタック保存
16 Def(
17   Tid(0x78f, "%0000078f"),
18   Attrs("[Attr("address", "0xC90"), Attr("insn", "stp x29, x30, [sp, #-0x80]!")]"),
19   Var("mem", Mem(0x40, 0x8)),
20   Store(Var("mem", Mem(0x40, 0x8)), PLUS(Var("SP", Imm(0x40)), Int(0xffffffffffffff80,
         0x40)), Var("FP", Imm(0x40)), LittleEndian(), 0x40)
21 )
22
23 # x30("LR") のスタック保存
24 Def(
25   Tid(0x794, "%00000794"),
26   Attrs("[Attr("address", "0xC90"), Attr("insn", "stp x29, x30, [sp, #-0x80]!")]"),
27   Var("mem", Mem(0x40, 0x8)),
28   Store(Var("mem", Mem(0x40, 0x8)), PLUS(Var("SP", Imm(0x40)), Int(0xffffffffffffff88,
         0x40)), Var("LR", Imm(0x40)), LittleEndian(), 0x40)
29 )
30
31 # sp の書き換え
32 Def(
33   Tid(0x799, "%00000799"),
34   Attrs("[Attr("address", "0xC90"), Attr("insn", "stp x29, x30, [sp, #-0x80]!")]"),
35   Var("SP", Imm(0x40)),
36   PLUS(Var("SP", Imm(0x40)), Int(0xffffffffffffff80, 0x40))
37 )
```

しかしながら，この BAP の中間表現には不具合があり，アンワインド情報合成に関わる命令に絞ると下記の情報が抜けている．

- stp reg1, reg2, [base, offset]
- str reg, [base, offset]!
- ldr reg , [base], offset

これらはアンワインド情報を合成するためには欠かせない命令であるため，抜けている情報を補完する必要がある．この補完には，pyelftools [7] でバイナリファイルを逆アセンブルしたコードの中から上記の 3 つの命令を抽出して，アンワインド情報合成時にそれに対応した処理を差し込むことによって解決している．

第5章　結果と考察

5.1　検査システム

5.1.1　結果

検査システムを動作させたときの例を抜粋して下記に記載する．この検査対象プログラムはC言語のコード（A.4 テストコード 1）をコンパイルしたバイナリファイルである．

ソースコード 5.1: A.4 テストコード 1 の.eh_frame 抜粋

```
000000b4 000000000000003c 000000b8 FDE cie=00000000 // fn1 関数
    pc=0000000000000834..0000000000000ad0
  DW_CFA_advance_loc: 4 to 0000000000000838
  DW_CFA_def_cfa_offset: 160
  DW_CFA_offset: r29 (x29) at cfa-160
  DW_CFA_offset: r30 (x30) at cfa-152
  DW_CFA_advance_loc: 4 to 000000000000083c
  DW_CFA_def_cfa_register: r29 (x29)
  DW_CFA_advance_loc: 24 to 0000000000000854
  DW_CFA_offset: r19 (x19) at cfa-144
  DW_CFA_offset: r20 (x20) at cfa-136
  DW_CFA_offset: r21 (x21) at cfa-128
  DW_CFA_offset: r22 (x22) at cfa-120
  DW_CFA_offset: r23 (x23) at cfa-112
  DW_CFA_offset: r24 (x24) at cfa-104
  DW_CFA_offset: r25 (x25) at cfa-96
  DW_CFA_offset: r26 (x26) at cfa-88
  DW_CFA_offset: r27 (x27) at cfa-80
  DW_CFA_advance_loc1: 632 to 0000000000000acc
  DW_CFA_restore: r30 (x30)
  DW_CFA_restore: r29 (x29)
  DW_CFA_restore: r27 (x27)
  DW_CFA_restore: r25 (x25)
  DW_CFA_restore: r26 (x26)
  DW_CFA_restore: r23 (x23)
  DW_CFA_restore: r24 (x24)
  DW_CFA_restore: r21 (x21)
  DW_CFA_restore: r22 (x22)
  DW_CFA_restore: r19 (x19)
  DW_CFA_restore: r20 (x20)
  DW_CFA_def_cfa: r31 (sp) ofs 0
  DW_CFA_nop
```

ソースコード 5.2: 検査システム結果抜粋（検査対象：A.4 テストコード 1）

```
~~~

0x0000aaaaaaaa0838 in fn1 ()
==========================================
['=> 0xaaaaaaaa0838 <fn1+4>:\tmov\tx29, sp\n']
==========================================
abstract_state:
[{'x30': '0x0000fffff7e609d4', 'x29': '0x0000ffffffffec60', 'x19': '0x0000aaaaaaaa0d50
    ', 'x20': '0x0000000000000000', 'x21': '0x0000aaaaaaaa0700', 'x22': '0
    x0000000000000000', 'x23': '0x0000000000000000', 'x24': '0x0000000000000000', 'x25
    ': '0x0000000000000000', 'x26': '0x0000000000000000', 'x27': '0x0000000000000000',
     'x28': '0x0'}, {'x30': '0xaaaaaaaa0cf0', 'x29': '0xffffffffebe0', 'x19': '0
    xffffffffeb20', 'x20': '0x20', 'x21': '0x0', 'x22': '0x20', 'x23': '0x0', 'x24':
     '0x1', 'x25': '0x0', 'x26': '0x1', 'x27': '0x0', 'x28': '0x0'}]

x30 comparison with a stack value > OK
x29 comparison with a stack value > OK
x19 store command check > OK
x20 store command check > OK
x21 store command check > OK
x22 store command check > OK
x23 store command check > OK
x24 store command check > OK
x25 store command check > OK
x26 store command check > OK
x27 store command check > OK
x28 store command check > OK

~~~~
~~~~

0x0000aaaaaaaa0840 in fn1 ()
==========================================
['=> 0xaaaaaaaa0840 <fn1+12>:\tstp\tx21, x22, [sp, #32]\n']
==========================================
abstract_state:
[{'x30': '0x0000fffff7e609d4', 'x29': '0x0000ffffffffec60', 'x19': '0x0000aaaaaaaa0d50
    ', 'x20': '0x0000000000000000', 'x21': '0x0000aaaaaaaa0700', 'x22': '0
    x0000000000000000', 'x23': '0x0000000000000000', 'x24': '0x0000000000000000', 'x25
    ': '0x0000000000000000', 'x26': '0x0000000000000000', 'x27': '0x0000000000000000',
     'x28': '0x0'}, {'x30': '0xaaaaaaaa0cf0', 'x29': '0xffffffffebe0', 'x19': '0
    xffffffffeb20', 'x20': '0x20', 'x21': '0x0', 'x22': '0x20', 'x23': '0x0', 'x24':
     '0x1', 'x25': '0x0', 'x26': '0x1', 'x27': '0x0', 'x28': '0x0'}]

x30 comparison with a stack value > OK
x29 comparison with a stack value > OK
not enough information in eh_frame
0xaaaaaaaa083c ['stp', 'x19', 'x20']
not enough information in eh_frame
0xaaaaaaaa083c ['stp', 'x19', 'x20']
x21 store command check > OK
x22 store command check > OK
x23 store command check > OK
```

```
x24 store command check > OK
x25 store command check > OK
x26 store command check > OK
x27 store command check > OK
x28 store command check > OK

~~~
~~~

0x0000aaaaaaaa0854 in fn1 ()
==========================================
['=> 0xaaaaaaaa0854 <fn1+32>:\tstr\tw0, [x29, #124]\n']
==========================================
abstract_state:
[{'x30': '0x0000fffff7e609d4', 'x29': '0x0000ffffffffec60', 'x19': '0x0000aaaaaaaa0d50
    ', 'x20': '0x0000000000000000', 'x21': '0x0000aaaaaaaa0700', 'x22': '0
    x0000000000000000', 'x23': '0x0000000000000000', 'x24': '0x0000000000000000', 'x25
    ': '0x0000000000000000', 'x26': '0x0000000000000000', 'x27': '0x0000000000000000',
     'x28': '0x0'}, {'x30': '0xaaaaaaaa0cf0', 'x29': '0xffffffffebe0', 'x19': '0
    xffffffffeb20', 'x20': '0x20', 'x21': '0x0', 'x22': '0x20', 'x23': '0x0', 'x24':
     '0x1', 'x25': '0x0', 'x26': '0x1', 'x27': '0x0', 'x28': '0x0'}]

x30 comparison with a stack value > OK
x29 comparison with a stack value > OK
x19 comparison with a stack value > OK
x20 comparison with a stack value > OK
x21 comparison with a stack value > OK
x22 comparison with a stack value > OK
x23 comparison with a stack value > OK
x24 comparison with a stack value > OK
x25 comparison with a stack value > OK
x26 comparison with a stack value > OK
x27 comparison with a stack value > OK
x28 need not to check

~~~
~~~

0x0000aaaaaaaa0acc in fn1 ()
==========================================
['=> 0xaaaaaaaa0acc <fn1+664>:\tret\n']
==========================================
abstract_state:
[{'x30': '0x0000fffff7e609d4', 'x29': '0x0000ffffffffec60', 'x19': '0x0000aaaaaaaa0d50
    ', 'x20': '0x0000000000000000', 'x21': '0x0000aaaaaaaa0700', 'x22': '0
    x0000000000000000', 'x23': '0x0000000000000000', 'x24': '0x0000000000000000', 'x25
    ': '0x0000000000000000', 'x26': '0x0000000000000000', 'x27': '0x0000000000000000',
     'x28': '0x0'}, {'x30': '0xaaaaaaaa0cf0', 'x29': '0xffffffffebe0', 'x19': '0
    xffffffffeb20', 'x20': '0x20', 'x21': '0x0', 'x22': '0x20', 'x23': '0x0', 'x24':
     '0x1', 'x25': '0x0', 'x26': '0x1', 'x27': '0x0', 'x28': '0x0'}]

x30 need not to check
x29 need not to check
x19 need not to check
```

```
x20 need not to check
x21 need not to check
x22 need not to check
x23 need not to check
x24 need not to check
x25 need not to check
x26 need not to check
x27 need not to check
x28 need not to check

ret function
Unwinging succesful
```

Rust 言語のコード（A.6 テストコード 3）をコンパイルしたときのアンワインド情報も DWARF 形式であるため，これを検査したところエラーを出力した．この出力は以下の通りである．

ソースコード 5.3: 検査システム結果抜粋（検査対象： A.6 テストコード 3）

```
エラー出力までの経過は省略

~~~

0x0000aaaaaaaaf914 80 in /build/rustc-svGMXs/rustc-1.51.0+dfsg1+llvm/library/std/src/
    sys/unix/mod.rs
==========================================
['=> 0xaaaaaaaaf914 <_ZN3std3sys4unix4init17h4524c7bdb602d7b5E+4>:\tmov\tw8, #0x1 \t//
     #1\n']
==========================================
abstract_state:
[{'x30': '0x0000fffff7df89d4', 'x29': '0xffffffffec10', 'x19': '0xaaaaaaaacf5a0', 'x20
    ': '0x0', 'x21': '0xaaaaaaaa4400', 'x22': '0x0', 'x23': '0x0', 'x24': '0x0', '
    x25': '0x0', 'x26': '0x0', 'x27': '0x0', 'x28': '0x0'}, {'x30': '0xaaaaaaaaa468c
    ', 'x29': '0xffffffffec10', 'x19': '0xaaaaaaaacf5a0', 'x20': '0x0', 'x21': '0
    xaaaaaaaa4400', 'x22': '0x0', 'x23': '0x0', 'x24': '0x0', 'x25': '0x0', 'x26':
    '0x0', 'x27': '0x0', 'x28': '0x0'}, {'x30': '0xaaaaaaaa47b8', 'x29': '0
    xffffffffec10', 'x19': '0xaaaaaaaacf5a0', 'x20': '0x0', 'x21': '0xaaaaaaaa4400', '
    x22': '0x0', 'x23': '0x0', 'x24': '0x0', 'x25': '0x0', 'x26': '0x0', 'x27': '0x0
    ', 'x28': '0x0'}, {'x30': '0xaaaaaaaad178', 'x29': '0xffffffffec10', 'x19': '0
    xffffffffebd8', 'x20': '0xaaaaaaae9a88', 'x21': '0xffffffffeaf0', 'x22': '0x0', '
    x23': '0x0', 'x24': '0x0', 'x25': '0x0', 'x26': '0x0', 'x27': '0x0', 'x28': '0x0
    '}]

error: comparison with a stack value > x30
real value: 0x0000fffff7ff5bc0
eh_frame value: 0xaaaaaaaad178
DecodedCallFrameTable(table=[{'pc': 63760, 'cfa': CFARule(reg=31, offset=0, expr=None
    )}, {'pc': 63764, 'cfa': CFARule(reg=31, offset=48, expr=None), 30: RegisterRule(
    OFFSET, -16)}], reg_order=[30])
```

5.1.2 考察

C 言語コードの検査結果からみる本システムの評価

まず，ソースコード 5.1 とソースコード 5.2 を比べる．ソースコード 5.1 によると，プログラムカウンタ 0000000000000838 の時点で x29，x30 レジスタの値ががスタック保存されている．ソースコード 5.2 の 3〜21 行目の検査結果を見ると，x30，x29 レジスタの値について検査が行われ，検査結果は OK となった．この比較は１ステップ実行ごとに行われており，4.1.1 節の機能 1 が実行できていることを確認した．

また，ソースコード 5.2，26〜46 行目の検査では，x19，x20 レジスタの値が実際の実行ではスタック保存されているのにも関わらず，.eh_frame ではその情報が記述されていないことを知らせる結果が出力された．これはエラーではなく，現時点で.eh_frame に反映されていないレジスタを表示している．本システムでは，検査中の関数において関数呼び出しおよび関数終了時に改めてエラーチェックを行うように設計しているため，ソースコード 5.18 行目，ソースコード 5.2 の 51〜69 行目の.eh_frame にレジスタ情報が反映されたときにはエラー出力をしていない．この結果から，4.1.1 節の機能 2 が実行できていることを確認した．

最後に，ソースコード 5.2，74〜95 行目では，スタック保存された値がレジスタに復帰し，その復帰したレジスタの値が関数呼び出し前の状態にアンワインド（巻き戻し）されているかを検査し，Unwinding successful という結果が得られているので，4.1.1 節の機能 3 が実行できていることを確認した．

以上のことから，本システムで設計した機能１，２，３，がすべて実行できており，本システムが設計通りに動作していると評価できた．

Rust コードの検査結果分析

ソースコード 5.3 の結果は，
/build/rustc-svGMXs/rustc-1.51.0+dfsg1+llvm/library/std/src/sys/unix/mod.rs ファイルに記載の _ZN3std3sys4unix4init17h4524c7bdb602d7b5E+4 関数におけるエラー出力である．これはプログラムカウンタ 0xf914 時点で，.eh_frame に記載される x30 レジスタの情報と実際の実行における x30 レジスタの情報が一致していないということが起きていた．このエラーを検出した関数のアセンブリコードと.eh_frame FDE 情報は以下の通りである．

ソースコード 5.4: エラー出力された関数 (A.6 テストコード 3) のアセンブリコード抜粋

```
1   0xf910 : sub sp, sp, #0x30
2   0xf914 : mov w8, #0x1 // #1
3   0xf918 : mov w9, #0x2 // #2
4   0xf91c : stp xzr, x8, [sp, #8]
5   0xf920 : stp x9, x30, [sp, #24]
6   0xf924 : add x0, sp, #0x8
7   0xf928 : mov w1, #0x3 // #3
8
9   ~~~
10
11  0xf9d4 : ldr x30, [sp, #32]
12  0xf9d8 : add sp, sp, #0x30
13  0xf9dc : ret
14
15  ~~~
```

表 5.1: エラー出力された関数の.eh_frame FDE の概念図

LOC	CFA	x30	x29	x19	⋯	x28
0xf910	sp+0	u	u	u	u	u
0xf914	sp+48	cfa−16	u	u	u	u

このエラーを検査アルゴリズム 4.1.1，機能 1 の処理で出力した．表 5.1 の 0xf914 で x30 レジスタの値がスタックに保存されているはずだが，ソースコード 5.4 では 0xf920 の命令で保存されるため，表 5.1 の 0xf914 の LOC は 0xf924 が正しいと思われる．しかし，この情報を用いたスタックアンワインドを考えると，LOC のズレはあれど，巻き戻し時点では正しいアドレスに x30 レジスタの値が保存されているため，正しくアンワインド可能である．したがって，このエラー出力された.eh_frame の箇所はバグではないと考えられる．

5.2 合成システム

5.2.1 結果

A.4 テストファイル 1 のアンワインド情報を合成した結果は以下の通りである．

ソースコード 5.5: 合成したアンワインド情報抜粋

```
1 // __libc_csu_init 関数
2 [{'pc': 3824, 'cfa': CFARule(reg=31, offset=0, expr=None)}, {'pc': 3828, 'cfa':
      CFARule(reg=31, offset=64, expr=None), 30: RegisterRule(OFFSET, -56), 29:
      RegisterRule(OFFSET, -64)}, {'pc': 3836, 'cfa': CFARule(reg=31, offset=64, expr=
      None), 30: RegisterRule(OFFSET, -56), 29: RegisterRule(OFFSET, -64), 19:
      RegisterRule(OFFSET, -48), 20: RegisterRule(OFFSET, -40)}, {'pc': 3848, 'cfa':
      CFARule(reg=31, offset=64, expr=None), 30: RegisterRule(OFFSET, -56), 29:
      RegisterRule(OFFSET, -64), 19: RegisterRule(OFFSET, -48), 20: RegisterRule(OFFSET,
       -40), 21: RegisterRule(OFFSET, -32), 22: RegisterRule(OFFSET, -24)}, {'pc':
      3868, 'cfa': CFARule(reg=31, offset=64, expr=None), 30: RegisterRule(OFFSET, -56)
      , 29: RegisterRule(OFFSET, -64), 19: RegisterRule(OFFSET, -48), 20: RegisterRule(
      OFFSET, -40), 21: RegisterRule(OFFSET, -32), 22: RegisterRule(OFFSET, -24), 23:
      RegisterRule(OFFSET, -16), 24: RegisterRule(OFFSET, -8)}, {'pc': 3932, 'cfa':
      CFARule(reg=31, offset=64, expr=None), 30: RegisterRule(OFFSET, -56), 29:
      RegisterRule(OFFSET, -64), 21: RegisterRule(OFFSET, -32), 22: RegisterRule(OFFSET,
       -24), 23: RegisterRule(OFFSET, -16), 24: RegisterRule(OFFSET, -8)}, {'pc':
      3936, 'cfa': CFARule(reg=31, offset=64, expr=None), 30: RegisterRule(OFFSET, -56)
      , 29: RegisterRule(OFFSET, -64), 23: RegisterRule(OFFSET, -16), 24: RegisterRule(
      OFFSET, -8)}, {'pc': 3940, 'cfa': CFARule(reg=31, offset=64, expr=None), 30:
      RegisterRule(OFFSET, -56), 29: RegisterRule(OFFSET, -64)}, {'pc': 3944, 'cfa':
      CFARule(reg=31, offset=0, expr=None)}],
3
4 // fn2 関数
5 [{'pc': 2420, 'cfa': CFARule(reg=31, offset=0, expr=None)}, {'pc': 2424, 'cfa':
      CFARule(reg=31, offset=48, expr=None), 30: RegisterRule(OFFSET, -40), 29:
      RegisterRule(OFFSET, -48)}, {'pc': 2524, 'cfa': CFARule(reg=31, offset=0, expr=
      None)}],
6
```

```
// fn1 関数
[{'pc': 2528, 'cfa': CFARule(reg=31, offset=0, expr=None)}, {'pc': 2532, 'cfa':
    CFARule(reg=31, offset=160, expr=None), 30: RegisterRule(OFFSET, -152), 29:
    RegisterRule(OFFSET, -160)}, {'pc': 2536, 'cfa': CFARule(reg=29, offset=160, expr
    =None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(OFFSET, -160)}, {'pc':
    2540, 'cfa': CFARule(reg=29, offset=160, expr=None), 30: RegisterRule(OFFSET,
    -152), 29: RegisterRule(OFFSET, -160), 19: RegisterRule(OFFSET, -144), 20:
    RegisterRule(OFFSET, -136)}, {'pc': 2544, 'cfa': CFARule(reg=29, offset=160, expr
    =None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(OFFSET, -160), 19:
    RegisterRule(OFFSET, -144), 20: RegisterRule(OFFSET, -136), 21: RegisterRule(
    OFFSET, -128), 22: RegisterRule(OFFSET, -120)}, {'pc': 2548, 'cfa': CFARule(reg
    =29, offset=160, expr=None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(
    OFFSET, -160), 19: RegisterRule(OFFSET, -144), 20: RegisterRule(OFFSET, -136),
    21: RegisterRule(OFFSET, -128), 22: RegisterRule(OFFSET, -120), 23: RegisterRule(
    OFFSET, -112), 24: RegisterRule(OFFSET, -104)}, {'pc': 2552, 'cfa': CFARule(reg
    =29, offset=160, expr=None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(
    OFFSET, -160), 19: RegisterRule(OFFSET, -144), 20: RegisterRule(OFFSET, -136),
    21: RegisterRule(OFFSET, -128), 22: RegisterRule(OFFSET, -120), 23: RegisterRule(
    OFFSET, -112), 24: RegisterRule(OFFSET, -104), 25: RegisterRule(OFFSET, -96), 26:
     RegisterRule(OFFSET, -88)}, {'pc': 2556, 'cfa': CFARule(reg=29, offset=160, expr
    =None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(OFFSET, -160), 19:
    RegisterRule(OFFSET, -144), 20: RegisterRule(OFFSET, -136), 21: RegisterRule(
    OFFSET, -128), 22: RegisterRule(OFFSET, -120), 23: RegisterRule(OFFSET, -112),
    24: RegisterRule(OFFSET, -104), 25: RegisterRule(OFFSET, -96), 26: RegisterRule(
    OFFSET, -88), 27: RegisterRule(OFFSET, -80)}, {'pc': 3172, 'cfa': CFARule(reg=29,
     offset=160, expr=None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(OFFSET,
     -160), 21: RegisterRule(OFFSET, -128), 22: RegisterRule(OFFSET, -120), 23:
    RegisterRule(OFFSET, -112), 24: RegisterRule(OFFSET, -104), 25: RegisterRule(
    OFFSET, -96), 26: RegisterRule(OFFSET, -88), 27: RegisterRule(OFFSET, -80)}, {'pc
    ': 3176, 'cfa': CFARule(reg=29, offset=160, expr=None), 30: RegisterRule(OFFSET,
    -152), 29: RegisterRule(OFFSET, -160), 23: RegisterRule(OFFSET, -112), 24:
    RegisterRule(OFFSET, -104), 25: RegisterRule(OFFSET, -96), 26: RegisterRule(OFFSET
    , -88), 27: RegisterRule(OFFSET, -80)}, {'pc': 3180, 'cfa': CFARule(reg=29,
    offset=160, expr=None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(OFFSET,
    -160), 25: RegisterRule(OFFSET, -96), 26: RegisterRule(OFFSET, -88), 27:
    RegisterRule(OFFSET, -80)}, {'pc': 3184, 'cfa': CFARule(reg=29, offset=160, expr=
    None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(OFFSET, -160), 27:
    RegisterRule(OFFSET, -80)}, {'pc': 3188, 'cfa': CFARule(reg=29, offset=160, expr=
    None), 30: RegisterRule(OFFSET, -152), 29: RegisterRule(OFFSET, -160)}, {'pc':
    3192, 'cfa': CFARule(reg=31, offset=0, expr=None)}],
```

以上それぞれの関数のアンワインド情報の表の形式にすると以下の通りである.

表 5.2: __libc_csu_init 関数の概念図

LOC	CFA	x30	x29	x19	x20	x21	x22	x23	x24
0xef0	sp+0	u	u	u	u	u	u	u	u
0xef4	sp+64	cfa−56	cfa−64	u	u	u	u	u	u
0xef8	sp+64	cfa−56	cfa−64	u	u	u	u	u	u
0xefc	sp+64	cfa−56	cfa−64	cfa−48	cfa−40	u	u	u	u
0xf08	sp+64	cfa−56	cfa−64	cfa−48	cfa−40	cfa−32	cfa−24	u	u
0xf1c	sp+64	cfa−56	cfa−64	cfa−48	cfa−40	cfa−32	cfa−24	cfa−16	cfa−8
0xf5c	sp+64	u	u	cfa−48	cfa−40	cfa−32	cfa−24	cfa−16	cfa−8
0xf60	sp+64	u	u	u	u	cfa−32	cfa−24	cfa−16	cfa−8
0xf64	sp+64	u	u	u	u	u	u	cfa−16	cfa−8
0xf68	sp+64	u	u	u	u	u	u	u	u

表 5.3: fn2 関数の概念図

LOC	CFA	x30	x29
0x974	sp+0	u	u
0x978	sp+48	cfa−40	cfa−48
0x9dc	sp+0	u	u

表 5.4: fn1 関数の概念図 (cfa を c と省略)

LOC	CFA	x30	x29	x19	x20	x21	x22	x23	x24	x25	x26	x27
0x9e0	sp+0	u	u	u	u	u	u	u	u	u	u	u
0x9e4	sp+128	c-120	c-128	u	u	u	u	u	u	u	u	u
0x9e4	x29+128	c-120	c-128	u	u	u	u	u	u	u	u	u
0x9ec	x29+128	c-120	c-128	c-112	c-104	u	u	u	u	u	u	u
0x9f0	x29+128	c-120	c-128	c-112	c-104	c-96	c-88	u	u	u	u	u
0x9f4	x29+128	c-120	c-128	c-112	c-104	c-96	c-88	c-80	c-72	u	u	u
0x9f8	x29+128	c-120	c-128	c-112	c-104	c-96	c-88	c-80	c-72	c-64	c-56	u
0x9fc	x29+128	c-120	c-128	c-112	c-104	c-96	c-88	c-80	c-72	c-64	c-56	c-48
0xc64	x29+128	c-120	c-128	u	u	c-96	c-88	c-80	c-72	c-64	c-56	c-48
0xc68	x29+128	c-120	c-128	u	u	u	u	c-80	c-72	c-64	c-56	c-48
0xc6c	x29+128	c-120	c-128	u	u	u	u	u	u	c-64	c-56	c-48
0x70	x29+128	c-120	c-128	u	u	u	u	u	u	u	u	c-48
0xc74	x29+128	c-120	c-128	u	u	u	u	u	u	u	u	u
0xc78	sp+0	u	u	u	u	u	u	u	u	u	u	u

また，A.5 テストファイル 2 で実行したところ合成に失敗した．合成失敗の原因については 5.2.2 節で分析する．

5.2.2 考察

本システムで合成したアンワインド情報とコンパイラで生成される.eh_frame の比較

5.2.1 節のアンワインド情報をコンパイラで合成された.eh_frame と比較する．以下は，コンパイラで合成されたテストファイルの.eh_frame を表の形式に直したものである．

表 5.5: コンパイラで合成した__libc_csu_init 関数のアンワインド情報概念図

LOC	CFA	x30	x29	x19	x20	x21	x22	x23	x24
0xef0	sp+0	u	u	u	u	u	u	u	u
0xef4	sp+64	cfa−56	cfa−64	u	u	u	u	u	u
0xef8	sp+64	cfa−56	cfa−64	u	u	u	u	u	u
0xefc	sp+64	cfa−56	cfa−64	cfa−48	cfa−40	u	u	u	u
0xf08	sp+64	cfa−56	cfa−64	cfa−48	cfa−40	cfa−32	cfa−24	u	u
0xf1c	sp+64	cfa−56	cfa−64	cfa−48	cfa−40	cfa−32	cfa−24	cfa−16	cfa−8
0xf68	sp+64	u	u	u	u	u	u	u	u

表 5.6: コンパイラで合成した fn2 関数のアンワインド情報概念図

LOC	CFA	x30	x29
0x974	sp+0	u	u
0x978	sp+48	cfa−40	cfa−48
0x9dc	sp+0	u	u

表 5.7: コンパイラで合成した fn1 関数のアンワインド情報概念図 (cfa を c と省略)

LOC	CFA	x30	x29	x19	x20	x21	x22	x23	x24	x25	x26	x27
0x9e0	sp+0	u	u	u	u	u	u	u	u	u	u	u
0x9e4	sp+128	c-120	c-128	u	u	u	u	u	u	u	u	u
0x9e4	x29+128	c-120	c-128	u	u	u	u	u	u	u	u	u
0x9ec	x29+128	c-120	c-128	c-112	c-104	u	u	u	u	u	u	u
0x9f0	x29+128	c-120	c-128	c-112	c-104	c-96	c-88	u	u	u	u	u
0x9f4	x29+128	c-120	c-128	c-112	c-104	c-96	c-88	c-80	c-72	u	u	u
0x9f8	x29+128	c-120	c-128	c-112	c-104	c-96	c-88	c-80	c-72	c-64	c-56	u
0x9fc	x29+128	c-120	c-128	c-112	c-104	c-96	c-88	c-80	c-72	c-64	c-56	c-48
0xc64	x29+128	c-120	c-128	u	u	c-96	c-88	c-80	c-72	c-64	c-56	c-48
0xc68	x29+128	c-120	c-128	u	u	u	u	c-80	c-72	c-64	c-56	c-48
0xc6c	x29+128	c-120	c-128	u	u	u	u	u	u	c-64	c-56	c-48
0x70	x29+128	c-120	c-128	u	u	u	u	u	u	u	u	c-48
0xc74	x29+128	c-120	c-128	u	u	u	u	u	u	u	u	u
0xc78	sp+0	u	u	u	u	u	u	u	u	u	u	u

比較したところ，本システムで合成されたアンワインド情報はコンパイラで合成された.eh_frame と異な

る点がある．

表 5.4 の callee-saved レジスタのスタック保存情報は 1 命令ずつ更新しているが，表 5.7 では，最後の x27 レジスタが保存された時点でそれまでのレジスタ保存をまとめて記述している．これはレジスタ復帰の際も同様である．これについて，表 5.5 では callee-saved レジスタのスタック保存がまとまって記述されてないことから，スタック保存の命令が連続して実行される表 5.7 はまとめて記述し，スタック保存の命令が連続して実行されていない表 5.5 はそれぞれの命令直後に記述されていると考えられる．また，表 5.4 では，すべての callee-saved レジスタの値がスタック保存されたときの LOC は 0xc98 だが，表 5.7 では，0xc9c となっている．これは，4.1.1 節で述べたように，実際の実行と.eh_frame 情報の LOC は多少のずれがあるということに原因がある．

これらの異なる点は，本システムによって合成されたアンワインド情報が誤っているのではなく，コンパイラで生成された.eh_frame よりも冗長な表現になっていることによるものだ．実際に，合成したアンワインド情報が誤っていないことを確かめるために，5.1 節の検査システムで検査したところ，エラー出力はされなかった．

合成に失敗したコード

本システムでアンワインド情報合成に失敗したコード (A.5 テストコード 2) を逆アセンブリしたコードを下に示す．

ソースコード 5.6: アンワインド情報合成に失敗したアセンブリコード抜粋

```
0000000000000b40 <main>:
 b40: mov x12, #0x1060
 b44: sub sp, sp, x12
 b48: stp x29, x30, [sp, #16]
 b4c: add x29, sp, #0x10
 b50: stp x19, x20, [sp, #32]
 b54: stp x21, x22, [sp, #48]
 b58: stp x23, x24, [sp, #64]
 b5c: stp x25, x26, [sp, #80]
 b60: str x27, [sp, #96]
 b64: str w0, [x29, #108]
 b68: str x1, [x29, #96]
```

本システムでは，sp の更新は sp+offset の形式を読み取って行い，offset はレジスタ値やメモリの値ではなく生の数値を前提としている．これ以外の sp の更新は追跡できないとして sp-index-mode であれば，x29-index-mode に切り替える設計にしている．ソースコード 5.6 では，LOC が 0xef4 のときの命令で offset がレジスタ値なので追跡できない．これが x29-index-mode であっても，x29 レジスタの値をスタック保存する前に追跡不能になっているため，x29 レジスタが cfa からどれだけ離れた位置に保存されるのか分からず，x29 にベースレジスタを切り替えることができないため，アンワインド情報の合成に失敗した．関数の始まりに sp が移動するのは AArch64 アーキテクチャの特徴であり，同じファイルを x86_64 アーキテクチャで逆アセンブリしても sp 更新ができない形式にはならなかった．この失敗を解決するために，offset にレジスタ値も含めて sp 更新を追跡することが考えられるが，offset として使われるレジスタが以前にどこで更新され，どんな値をとるのかという前方解析が必要になり，そのレジスタが他のレジスタ値やメモリの値を使用した計算結果になっている場合，さらにそのレジスタやメモリの前方解析が必要になると考えられる．

第6章　まとめ

本研究では，AArch64 アーキテクチャの命令セットと呼び出し規則に合わせた.eh_frame のバグの存在を検査するシステム，バイナリファイルからアンワインド情報を合成するシステムを設計，開発した．検査システムでは，既存研究である x86_64 アーキテクチャ用の検査システムで検査可能なスタックポインタ更新のバグ検査機能を拡張し，各 callee-saved レジスタが.eh_frame の記載通りにスタック保存されているか検査する機能とした．加えて，.eh_frame に記載されていないレジスタ保存を行ってはいないか，保存した値が元のレジスタに復帰しているかということを検査する機能も追加した．また，本システムで追加した機能におけるバグ検査の考え方は， [2] にも追加実装可能であり，検査するバグの対象範囲の拡大が期待される．合成システムでは，AArch64 でみられる命令セットや呼び出し規約を理解し，実際のコードと整合性が取れたアンワインド情報の合成を行った．一部のコードに対応していないため，随時アップデートを行う．この検査システムと合成システムによって，.eh_frame のバグのある FDE を特定，その FDE を合成システムによって合成されたアンワインド情報に差し替えることが可能になる．合成されたアンワインド情報のバイナリファイルへの組み込み，.eh_frame を活用した安全なシステム設計が今後の課題である．

謝辞

本研究を遂行するにあたり，本学の宮地充子教授から熱心な御指導御鞭撻を賜りました. また，本論文 の構成に関して有益な御指導を数多く頂きました，宮地充子教授に深く感謝いたします．高野祐輝准教授には，本研究の基礎知識から実装まで多大な御指導を頂き，行き詰まった際には心強く励ましてくださったおかげで本論文執筆まで至りました．心より感謝いたします．また，母語ではないにも関わらず，本論文を査読して頂いた Sai Veerya Mahadevan さん，Yaoan Jin さんに深く感謝いたします．最後に様々な面で支援して頂いた宮地研究室の諸氏に感謝いたします.

参考文献

[1] Arm. Procedure Call Standard for the Arm® 64-bit Architecture, 2018. https://developer.arm.com/documentation/ihi0055/latest.

[2] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. Reliable and Fast DWARF-Based Stack Unwinding. volume 3, pages 5–12, New York, NY, USA, oct 2019. Association for Computing Machinery.

[3] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, November 2020.

[4] Inc Free Software Foundation. Debugging with GDB, 1988. https://sourceware.org/gdb/current/onlinedocs/gdb/.

[5] Inc Free Software Foundation. Using the GNU Compiler Collection (GCC) Optioin Summary, 1988. https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc/Option-Summary.html#Option-Summary.

[6] Ivan Gotovchits. Binary Analysis Platform, 2022. https://github.com/BinaryAnalysisPlatform/bap.

[7] Brendan Hines. pyelftools, 2022. https://github.com/eliben/pyelftools.

[8] intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, 2021. https://cdrdv2.intel.com/v1/dl/getContent/671436.

[9] DWARF Debugging Information Format Workgroup. *DWARF Debugging Information Format Version 3*, 2005. https://dwarfstd.org/doc/Dwarf3.pdf.

[10] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[11] 大崎　博之. **独習アセンブラ新版**. 株式会社翔泳社, 2022.

[12] 高野　祐輝. **並行プログラミング入門**. オライリー・ジャパン, 2021.

付 録A　プログラム

ソースコード A.1: 検査システムのコード (validation.py)

```
import re
import sys
sys.path.append('PATH')
import synthesis
from elftools.elf.elffile import ELFFile
from elftools.dwarf.callframe import ZERO
import gdb

#input dwarf info
def process_file(filename):
    print('Processing file:', filename)
    with open(filename, 'rb') as f:
        elffile = ELFFile(f)

        if not elffile.has_dwarf_info():
            print(' file has no DWARF info')
            return

        dwarfinfo = elffile.get_dwarf_info()
        fdes = dwarfinfo.EH_CFI_entries()

    return fdes

#decode table
def get_decode_table(frame_table):
    decode = frame_table.get_decoded()
    return decode.table

#get the top address list of fdes
def make_pc_list(eh_frame):
    global mode

    pc_list = []
    if mode == 'syn':
        for table in eh_frame:
            table_init = table[0]
            pc_list.append(table_init['pc'])
    else:
        for frame_table in eh_frame:
            if type(frame_table) != ZERO:
                table = get_decode_table(frame_table)
                table_init = table[0]
                pc_list.append(table_init['pc'])
    return pc_list
```

```
45
46 #get the num of fde included pc
47 def check_pc_block(pc_list, pc):
48     tmp_num = 0
49     for num in range(len(pc_list)):
50         if pc_list[num] <= pc:
51             if pc_list[tmp_num] < pc_list[num]:
52                 tmp_num = num
53     return tmp_num
54
55 #get an offset information of pc && reg
56 def get_offset(frame_table, pc, reg):
57     global mode
58     reg = int(reg)
59
60     if mode == 'syn':
61         table = frame_table
62     else:
63         if type(frame_table) != ZERO:
64             table = get_decode_table(frame_table)
65         else:
66             print('error: get_offset ZERO')
67
68     for dic in table:
69         if dic['pc'] <= pc:
70             state_dic = dic
71         else:
72             break
73
74     if state_dic.get('cfa') and state_dic.get(reg):
75         return state_dic['cfa'].offset, state_dic['cfa'].reg, state_dic[reg].arg
76
77     return None, None, None
78
79 #get current regs information
80 def get_info():
81     regs = gdb.execute('i r', to_string=True)
82     regs_list = regs.split('\n')
83     info = {}
84     for i in range(len(regs_list)-1):
85         line = regs_list[i].split()
86         info[line[0]] = line[1]
87     return info
88
89 #get current instr information
90 def get_next_instr():
91     assemble = gdb.execute('x/1i $pc',to_string = True)
92     assemble_list = assemble.split(':')
93     assemble_pcside = assemble_list[0].split()
94     assemble_instrside = assemble_list[1].split('[')
95     assemble_instr_operand = re.split('[ \],\t#!\n]', assemble_instrside[0])
96     instr = [a for a in assemble_instr_operand if a != '']
97     pc = assemble_pcside[1]
98     assemble_list = re.split(':\n\t',assemble)
```

```
99      print('=========================================')
100     print(assemble_list)
101     print('=========================================')
102     return pc,instr
103
104 #form an address
105 def attach_mask(addr):
106     mask = "0xaaaaaaaa0000"
107     mask = int(mask,0)
108     addr = int(addr,0)
109     addr = addr - mask
110     return addr
111
112 #from contents
113 def get_contents(content):
114     content_list = content.split()
115     addr = '0x'
116     for i in content_list[8:0:-1]:
117         addr = addr + i[2:4]
118     return addr
119
120 #check loaded regs
121 def check_dict(row, info, current_load_regs):
122     fault_regs = []
123     for reg in current_load_regs:
124         if row[reg] != info[reg]:
125             fault_regs.append(reg)
126     if any(fault_regs):
127         return fault_regs
128     else:
129         return True
130
131 #get operands
132 def get_operand(instr_list):
133     ope = []
134     if instr_list[0] == 'str':
135         ope.append(instr_list[1])
136     elif instr_list[0] == 'stp':
137         ope.append(instr_list[1])
138         ope.append(instr_list[2])
139     else:
140         print('error: getOperand')
141         exit()
142     return ope
143
144 #check error_regs
145 def check_error(error):
146     for reg in error:
147         if error[reg] != None:
148             return True
149     return False
150
151
152
```

```
153
154 mode = 'val' #or syn < validate a synthesis stack unwinding information
155 filename = 'test'
156 gdb.execute('set pagination off')
157
158 if mode == 'syn':
159     eh_frame = synthesis.synmain(filename)
160 else:
161     eh_frame = process_file(filename)
162
163 abstract_state = []
164 pc_list = make_pc_list(eh_frame)
165 print(pc_list)
166 gdb.execute('file ' + filename)
167 gdb.execute('start')
168 pc, instr = get_next_instr()
169 info = get_info()
170 mpc = attach_mask(pc)
171 list_num = check_pc_block(pc_list, mpc)
172 check_frag = False
173 error_regs = {'x30': None, 'x29': None, 'x19': None, 'x20': None, 'x21': None, 'x22':
        None, 'x23': None, 'x24': None, 'x25': None, 'x26': None, 'x27': None, 'x28':
       None}
174 regs_info = {'x30': info['x30'], 'x29': info['x29'], 'x19': info['x19'], 'x20': info
       ['x20'], 'x21': info['x21'], 'x22': info['x22'], 'x23': info['x23'], 'x24': info
       ['x24'], 'x25': info['x25'], 'x26': info['x26'], 'x27': info['x27'], 'x28': info
       ['x28']}
175 store_regs = []
176 current_store_regs = []
177 load_regs = []
178
179 #initial_process
180 for reg in regs_info:
181     cfa_offset, index_reg, reg_offset = get_offset(eh_frame[list_num], mpc, reg[1::])
182     if cfa_offset != None and reg_offset != None:
183         if index_reg == 31:
184             index = int(info['sp'], 0)
185         elif index_reg == 29:
186             index = int(info['x29'], 0)
187         else:
188             print('error reg')
189             print(reg)
190             exit()
191         reg_address = index + cfa_offset + reg_offset
192         reg_address = hex(reg_address)
193         reg_address_info_pre = gdb.execute('x/8xb '+ reg_address ,to_string=True)
194         reg_address_info = get_contents(reg_address_info_pre)
195         regs_info[reg] = reg_address_info
196         current_store_regs.append(reg)
197 abstract_state.append(regs_info)
198 store_regs.append(current_store_regs)
199 current_store_regs = []
200 pre_pc, pre_instr = pc, instr
201 gdb.execute('stepi')
```

```
202
203 while True:
204     pc, instr = get_next_instr()
205     info = get_info()
206     regs_info = {'x30': info['x30'], 'x29': info['x29'], 'x19': info['x19'], 'x20':
            info['x20'], 'x21': info['x21'], 'x22': info['x22'], 'x23': info['x23'], 'x24
            ': info['x24'], 'x25': info['x25'], 'x26': info['x26'], 'x27': info['x27'], '
            x28': info['x28']}
207     mpc = attach_mask(pc)
208     list_num = check_pc_block(pc_list, mpc)
209     print('abstract_state:')
210     print(abstract_state)
211     print()
212
213     #all regs of regs_info.keys()
214     for reg in regs_info:
215         cfa_offset, index_reg, reg_offset = get_offset(eh_frame[list_num], mpc, reg
                [1::])
216
217         #if reg information exists in fde:
218         if cfa_offset != None and reg_offset != None:
219             current_store_regs.append(reg)
220             if index_reg == 31:
221                 index = int(info['sp'], 0)
222             elif index_reg == 29:
223                 index = int(info['x29'], 0)
224             else:
225                 print('error reg')
226                 print(reg)
227                 exit()
228             reg_address = index + cfa_offset + reg_offset
229             reg_address = hex(reg_address)
230             reg_address_info_pre = gdb.execute('x/8xb '+ reg_address ,to_string=True)
231             reg_address_info = get_contents(reg_address_info_pre)
232
233             #fix error_regs
234             if error_regs[reg] != None:
235                 error_regs[reg] = None
236
237             if any(abstract_state):
238
239                 #validate real   executionand eh_frame information (function1)
240                 if int(reg_address_info, 0) != int(abstract_state[-1][reg],0):
241                     print('error: comparison with a stack value > ' + reg)
242                     print('real value: '+ reg_address_info)
243                     print('eh_frame value: '+ abstract_state[-1][reg])
244                     print(eh_frame[list_num].get_decoded())
245                     exit()
246                 else:
247                     print(reg + ' comparison with a stack value > OK')
248
249         else:
250             #push loaded regs
251             if reg in current_store_regs:
```

```
252                 load_regs.append(reg)
253
254             #validate real   executionand eh_frame information (function2)
255             if pre_instr[0] == 'str':
256                 pre_operand = get_operand(pre_instr)
257                 if reg in pre_operand:
258                     error_regs[pre_operand[0]] = pre_pc
259                     print('not enough information in eh_frame')
260                     print(pre_pc,pre_instr)
261                 else:
262                     print(reg + ' store command check > OK')
263
264             elif pre_instr[0] == 'stp':
265                 pre_operand = get_operand(pre_instr)
266                 if reg in pre_operand:
267                     error_regs[pre_operand[0]] = pre_pc
268                     error_regs[pre_operand[1]] = pre_pc
269                     print('not enough information in eh_frame')
270                     print(pre_pc, pre_instr)
271                 else:
272                     print(reg + ' store command check > OK')
273             else:
274                 print(reg + ' need not to check')
275
276     if instr[0] == 'bl':
277         #check error_regs(function2)
278         if check_error(error_regs):
279             print('error: not enough information in eh_frame')
280             print(error_regs)
281         print('bl function')
282         gdb.execute('stepi')
283         info = get_info()
284         abstract_state.append({'x30': info['x30'], 'x29': info['x29'], 'x19': info['
                x19'], 'x20': info['x20'], 'x21': info['x21'], 'x22': info['x22'], 'x23':
                 info['x23'], 'x24': info['x24'], 'x25': info['x25'], 'x26': info['x26'],
                 'x27': info['x27'], 'x28': info['x28']})
285         store_regs.append(current_store_regs)
286         current_store_regs = []
287     elif instr[0] == 'blr':
288         #check error_regs(function2)
289         if check_error(error_regs):
290             print('error: not enough information in eh_frame')
291             print(error_regs)
292         print('bl function')
293         gdb.execute('stepi')
294         info = get_info()
295         abstract_state.append({'x30': info['x30'], 'x29': info['x29'], 'x19': info['
                x19'], 'x20': info['x20'], 'x21': info['x21'], 'x22': info['x22'], 'x23':
                 info['x23'], 'x24': info['x24'], 'x25': info['x25'], 'x26': info['x26'],
                 'x27': info['x27'], 'x28': info['x28']})
296         store_regs.append(current_store_regs)
297         current_store_regs = []
298     elif instr[0] == 'ret':
299         #check error_regs(function2)
```

```
300         if check_error(error_regs):
301             print('error: not enough information in eh_frame')
302             print(error_regs)
303         print('ret function')
304         row = abstract_state.pop()
305 
306         #check loaded regs (function3)
307         if check_dict(row, info, load_regs):
308             print('Unwinging succesful')
309             if not abstract_state:
310                 print('All Unwinding Successful')
311                 break
312             current_store_regs = store_regs[-1]
313             load_regs = []
314             gdb.execute('stepi')
315 
316         else:
317             print('Unwinding False')
318             print({'x30': info['x30'], 'x29': info['x29'], 'x19': info['x19'], 'x20':
                     info['x20'], 'x21': info['x21'], 'x22': info['x22'], 'x23': info['x23
                    '], 'x24': info['x24'], 'x25': info['x25'], 'x26': info['x26'], 'x27
                    ': info['x27'], 'x28': info['x28']})
319             break
320     else:
321         gdb.execute('stepi')
322     pre_pc, pre_instr = pc, instr
323 gdb.execute('q')
```

ソースコード A.2: bap の欠けた命令を直接読み込むコード ($\mathrm{read}_a ssemble.pv$)

```
1  import sys
2  from elftools.elf.elffile import ELFFile
3  from capstone import *
4  import re
5 
6  #get opperand
7  def process_file(filename):
8      print('Processing file:', filename)
9      with open(filename, 'rb') as f:
10         elf = ELFFile(f)
11         code = elf.get_section_by_name('.text')
12         ops = code.data()
13         addr = code['sh_addr']
14         md = Cs(CS_ARCH_ARM64, CS_MODE_ARM)
15         callee_list = {'x19': 'X19', 'x20': 'X20', 'x21': 'X21', 'x22': 'X22', 'x23':
                'X23', 'x24': 'X24', 'x25': 'X25', 'x26': 'X26', 'x27': 'X27', 'x28': '
               X28', 'x29': 'FP', 'x30': 'LR'}
16         dict = {}
17         for i in md.disasm(ops, addr):
18             if i.mnemonic == 'stp':
19                 ope = i.op_str.split('[')
20                 ope_regs_list = re.split('[ ,\t]', ope[0])
21                 ope_regs = [a for a in ope_regs_list if a != '']
```

```
22                 if ope[1][-1] == ']':
23                     ope_offset_list = re.split('[ ,\]\t#]', ope[1])
24                     ope_offset = [a for a in ope_offset_list if a != '']
25                     if ope_regs[0] in callee_list or ope_regs[1] in callee_list:
26                         dict[i.address] = {'instr': i.mnemonic, 'regs': [callee_list[
                               ope_regs[0]],callee_list[ope_regs[1]]] , 'offset':
                               ope_offset}
27             elif i.mnemonic == 'str':
28                 ope = i.op_str.split('[')
29                 ope_regs_list = re.split('[ ,\t]', ope[0])
30                 ope_regs = [a for a in ope_regs_list if a != '']
31                 if ope[1][-1] == '!':
32                     ope_offset_list = re.split('[ \],\t#!]', ope[1])
33                     ope_offset = [a for a in ope_offset_list if a != '']
34                     if ope_regs[0] in callee_list:
35                         dict[i.address] = {'instr': i.mnemonic, 'regs': [callee_list[
                               ope_regs[0]]], 'offset': ope_offset}
36             elif i.mnemonic == 'ldr':
37                 ope = i.op_str.split('[')
38                 ope_regs_list = re.split('[ ,\t]', ope[0])
39                 ope_regs = [a for a in ope_regs_list if a != '']
40                 if '],' in ope[1]:
41                     ope_offset_list = re.split('[ \],\t#!]', ope[1])
42                     ope_offset = [a for a in ope_offset_list if a != '']
43                     if ope_regs[0] in callee_list:
44                         dict[i.address] = {'instr': i.mnemonic, 'regs': [callee_list[
                               ope_regs[0]]], 'offset': ope_offset}
45         print(dict)
46         return dict
47
48
49
50 if __name__ == '__main__':
51     if sys.argv[1] == '--test':
52         for filename in sys.argv[2:]:
53             instr_dict = process_file(filename)
```

ソースコード A.3: アンワインド情報を検査アルゴリズムが認識可能な形式に直すコード (exchange.pv)

```
1  from elftools.elf.elffile import ELFFile
2  from elftools.dwarf.callframe import ZERO
3  from elftools.dwarf.callframe import RegisterRule
4  from elftools.dwarf.callframe import CFARule
5  from elftools.dwarf.callframe import ZERO
6  from elftools.elf.elffile import ELFFile
7
8  def set_register(return_row, table, row_pc):
9      if table[row_pc]["LR"] != None:
10         reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["LR"])
11         return_row[30] = reg_data
12     if table[row_pc]["FP"] != None:
13         reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["FP"])
14         return_row[29] = reg_data
```

```
15      if table[row_pc]["X19"] != None:
16          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X19"])
17          return_row[19] = reg_data
18      if table[row_pc]["X20"] != None:
19          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X20"])
20          return_row[20] = reg_data
21      if table[row_pc]["X21"] != None:
22          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X21"])
23          return_row[21] = reg_data
24      if table[row_pc]["X22"] != None:
25          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X22"])
26          return_row[22] = reg_data
27      if table[row_pc]["X23"] != None:
28          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X23"])
29          return_row[23] = reg_data
30      if table[row_pc]["X24"] != None:
31          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X24"])
32          return_row[24] = reg_data
33      if table[row_pc]["X25"] != None:
34          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X25"])
35          return_row[25] = reg_data
36      if table[row_pc]["X26"] != None:
37          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X26"])
38          return_row[26] = reg_data
39      if table[row_pc]["X27"] != None:
40          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X27"])
41          return_row[27] = reg_data
42      if table[row_pc]["X28"] != None:
43          reg_data = RegisterRule(RegisterRule.OFFSET, table[row_pc]["X28"])
44          return_row[28] = reg_data
45
46      return return_row
47
48  def set_CFA(table, row_pc):
49      cfa_data = CFARule()
50      if table[row_pc]['base'] == 'SP':
51          base = 31
52      elif table[row_pc]['base'] == 'FP':
53          base = 29
54      else:
55          print('error: exchange.py setCFA')
56          exit()
57      cfa_data = CFARule(base, table[row_pc]['CFA'], None)
58      return cfa_data
59
60  def exchange_table(eh_frame_list):
61      eh_frame = []
62      for table in eh_frame_list:
63          return_table = []
64
65          for row_pc in table:
66              return_row = {}
67              return_row['pc'] = int(row_pc, 0)
68              return_row['cfa'] = set_CFA(table, row_pc)
```

```
69              return_row = set_register(return_row, table, row_pc)
70              return_table.append(return_row)
71
72          eh_frame.append(return_table)
73      return eh_frame
```

ソースコード A.4: テストコード 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  short a, b;
5  char d;
6
7  int fn4(int *v){
8      return *v;
9  }
10
11 int fn3(){
12     return 3;
13 }
14
15 int fn2(int j){
16     int *i;
17
18     i = (int*)malloc(sizeof(int));
19     if (i == NULL) {
20         printf("error: malloc");
21         return -1;
22     }
23
24     scanf("%n", i);
25
26     int r = *i;
27
28     free(i);
29     return 2 + j;
30 }
31
32 int fn1(int b, int b1, int b2, int b3, int b4, int b5, int b6, int b7, int b8, int b9
       , int b10, int b11, int b12, int b13, int b14, int b15, int b16, int b17, int b18,
        int b19, int b20, int b21, int b22, int b23, int b24, int b25, int b26, int b27,
       int b28, int b29, int b30){
33     float c;
34
35     b = b + b1 + b2 + b3 + b4 + b5 + b6 + b7 + b8 + b9 + b10 + b11 + b12 + b13 + b14
            + b15 + b16 + b17 + b18 + b19 + b20 + b21 + b22 + b23 + b24 + b25 + b26 +
           b27 + b28 + b29 + b30;
36     for (int i = 1; i < 2; i++) {
37         int v[i];
38         int *p;
39         c = c + fn3();
40     }
```

```
41      b = b + c;
42      return b;
43  }
44
45  void main(){
46      a=4;
47      for (int i = 1; i < 2; i++) {
48          int v[i];
49          d = d + fn3();
50      }
51      a=fn1(8,
            1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30)
            ;
52  }
```

ソースコード A.5: テストコード 2

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4   #include <time.h>
5   #include <string.h>
6
7   double fn2(double x, double y, double a, double b){
8           return x + y + a + b;
9   }
10
11  int fn1(double *p, int size, double x, double y){
12          double d;
13          double d_dash;
14          int num=0;
15
16          d = fn2(x, y, p[0], p[1]);
17          for(int i=1; i<size; i++){
18                  d_dash = fn2(x, y, p[i*2], p[i*2+1]);
19                  if(d > d_dash){
20                          d = d_dash;
21                          num=i;
22                  }
23          }
24          return num;
25  }
26
27  int main(int argc, char *argv[])
28  {
29          FILE *input_file;
30          int input;
31          int count=0;
32          double data[200][2];
33          int label[200];
34          int i=0;
35          int j=0;
36
```

```
37          input_file = fopen(argv[1],"r");
38          if(input_file==NULL){
39                  printf("input error\n");
40                  return -1;
41          }
42          while( (input=fscanf(input_file, "%lf,%lf", &data[count][0], &data[count][1]))
                 != EOF){
43                  count++;
44          }
45
46          double *p;
47          int n;
48
49          printf("Input a k : ");
50          scanf("%d", &n);
51
52          int N = 2*n;
53          int array[n];
54
55          p = (double *)malloc(N *sizeof(double));
56
57          for(i=0; i<200; i++){
58                  label[i] = fn1(p, n, data[i][0], data[i][1]);
59          }
60
61          free(p);
62          return 0;
63  }
```

ソースコード A.6: テストコード 3

```
1  fn fnc2(a: i32, b: i32) -> i32{
2      return a*b
3  }
4
5  fn fnc1(a: i32, b: i32, c: i32) -> i32 {
6      let x;
7      x = fnc2(a,c);
8      return a+b+x;
9  }
10
11 fn main() {
12     let a = 1;
13     let b = 2;
14     let c = 3;
15     let _d;
16     _d = fnc1(a,b,c);
17 }
```

付 録B　本研究に対する発表論文

本研究に対する発表論文を以下に列挙する．

- 川口　哲弘，高野　祐輝，宮地　充子．AArch64 の呼び出し規約に則ったアンワインド情報検査システムの開発In ICSS: *IPSJ-SPT*. 2022 年 3 月発表予定