修士学位論文

題目

Unix2038 年問題に関するソースコード修正支援ツールの作成

指導教員
肥後 芳樹 教授

報告者
水上 陽向

令和 5 年 2 月 1 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

令和 4 年度 修士学位論文

Unix2038 年問題に関するソースコード修正支援ツールの作成

水上 陽向

内容梗概

Unix2038 年問題とは，Unix ベースのシステムにおいて，2038 年に時刻情報を扱う time_t 型の桁あふれが発生することに起因する問題である．

この Unix2038 年問題に対するリスクの調査を行った研究は，特定環境下での不具合のリスクについて調査したものしか確認できない．また，Unix2038 年問題への対応の低コスト化を図る研究は複数存在するが，対応の実行は人力で行う必要がある．現状では 2038 年問題への対応には，小さくない労力が必要となっている．

本研究では Unix2038 年問題への対応をより容易にすることを目指し，先行研究の手法に基づくソースコード修正を支援するツールの作成を行う．また，事前調査として，OSS を対象とした Unix2038 年問題の潜在リスクの調査を行い，Unix2038 年問題への対応を容易化することの意義を示す．

事前調査の結果から，Unix2038 年問題による不具合は，多くのソフトウェアで発生する危険があり，特に time_t 型が符号付 32bit である環境においては，致命的な問題となる可能性があることがわかった．

作成したツールは，先行研究により提案された，システム内で扱う Unix 時刻の起算点を変更し，オーバーフローの発生を先送りする手法に基づく，Unix2038 年問題への対応のためのソースコード修正を支援するものである．

評価の結果，作成ツールは採用手法に基づき，ソースコード中の修正が必要な箇所を高い再現率で特定し，適切な修正案を提示することができることが示された．Unix2038 年問題の対応のために修正が必要なソースコード中の箇所の特定と，それらに対する修正の実行の作業の省力化が可能である．

主な用語

2038 年問題
Unix 時刻
ソースコード修正
オーバーフロー

目 次

1　まえがき

Unix2038 年問題とは，Unix ベースのシステムにおいて，2038 年に時刻情報を扱う変数の桁あふれが発生することに起因する問題である．

Unix ベースシステムにおける時刻情報は，1970 年 1 月 1 日を起算点とする，Unix 時刻が用いられている [1]．この Unix 時刻を扱うための変数型として，time_t 型が用意されている [1]．time_t 型は，起算点からの経過秒数を保持する整数型であり，そのデータサイズはシステム依存とされている．1970 年 1 月 1 日からの経過秒数は，2038 年 1 月 19 日に符号付 32bit によって表せる最大値を超えるため，time_t 型を符号付 32bit で定義しているシステムでは，時刻の解釈が誤ったものとなり，不具合の発生につながる．

この Unix2038 年問題に対するリスクの調査を行った研究は，特定環境下での不具合のリスクについて調査した Suzuki ら [2] のものしか確認できない．また，大江ら [3, 4]，Okabe ら [5]，著者ら [6] は，Unix2038 年問題への対応の低コスト化を図る研究を行っているが，対応の実行は人力で行う必要がある．このことから，現状では 2038 年問題への対応には，小さくない労力が必要となっている．

これらを踏まえ，本研究では Unix2038 年問題への対応をより容易にすることを目指す．この目的のため，大江ら [3] の手法に基づくソースコード修正を支援するツールの作成を行う．また，事前調査として，OSS を対象とした Unix2038 年問題の潜在リスクの調査を行い，Unix2038 年問題への対応を容易化することの意義を示す．

事前調査では，GitHub[7] 上に存在する OSS を対象とし，符号付 32bit の time_t 型のオーバーフローによる不具合と，Suzuki ら [2] によって指摘された，time_t 型の 64bit 化による不具合に繋がるソースコード中の潜在リスクの実態を調査した．ソースコードからリスクとなる箇所を検出するためには，ソースコード解析ツール Understand[8] を用いた静的ソースコード解析を行った．

調査の結果，対象とした 172 のプロジェクト中，1 つ以上のリスクが検出されたプロジェクトは 108 個あり，対象のうち 62.8%に相当した．検出されたリスクの内訳は，符号付 32bit の time_t 型のオーバーフローに関するリスクが全体の 90%以上を占めていた．これらの結果から，Unix2038 年問題による不具合は，多くのソフトウェアで発生する危険があり，特に time_t 型が符号付 32bit である環境においては，致命的な問題となる可能性があることがわかった．

こうした事前調査の結果を踏まえ，本研究で作成したツールは，大江ら [3, 4] により提案された手法に基づく，Unix2038 年問題への対応のためのソースコード修正を支援するものである．この手法では，システム内で扱う Unix 時刻の起算点を変更し，オーバーフローの発生を先送りすることで，Unix2038 年問題への対応を行う．

作成ツールは，ソースコード解析ツール Understand[8] を用いて修正対象の C 言語ソースファイルの静的解析を行い，修正が必要となる箇所を特定する．その後，それらを分類し，修正案を作成する．自動での修正案作成が困難なものは，その位置を通知する．

作成ツールの評価として，FreeBSD[9] 12.1-RELEASE のソースコードを対象に作成ツールの実行を行った．評価の結果から，作成ツールは大江ら [4] の Unix2038 年問題への対応手法に基づき，ソースコード中の修正が必要な箇所を高い再現率で特定し，適切な修正案を提示することができることが示された．これを利用することにより，従来は全て人力で行う必要があった，Unix2038 年問題の対応のために修正が必要なソースコード中の箇所の特定と，それらに対する修正の実行の作業の省力化が可能である．

以降，2 節では，研究の背景となる Unix2038 年問題についてと，その類似の問題について，また関連研究やその課題について述べる．3 節では，事前調査として行った，OSS を対象とした Unix2038 年問題に対する潜在リスク調査について述べる．4 節では，Unix2038 年問題に対するソースコード修正支援ツールの作成について述べる．5 節では，今後の課題について述べる．最後に，6 節に，本研究のまとめを述べている．

2　背景

2.1　2038 年問題

Unix2038 年問題（以降，単に 2038 年問題と呼ぶ）とは，Unix ベースのシステムにおいて，2038 年に時刻情報を扱う変数の桁あふれが発生することに起因する問題である．本節では，この問題について述べる．

Unix ベースシステムにおける時刻情報は，1970 年 1 月 1 日を起算点とする，Unix 時刻 [1] が用いられている．Unix 時刻を扱うための変数型として，time_t 型が用意されている [1]．time_t 型は，起算点からの経過秒数を保持する整数型であり，そのデータサイズはシステム依存とされている．

図 1 に示すように，1970 年 1 月 1 日からの経過秒数は，2038 年 1 月 19 日に符号付 32bit によって表せる最大値を超える．これにより，time_t 型を符号付 32bit で定義しているシステムでは，時刻の解釈が誤ったものとなり，不具合の発生につながる．具体的には，以下のような不具合が想定される．

- 時刻が負値となることによる，比較結果の大小逆転
- 時刻が負値となることによる，エラーの発生
- 時刻情報の不整合による，通信エラー

現在時刻を 2038 年以降に変更した状況において，実際にソフトウェアが不具合を起こした事例が存在する [10]．

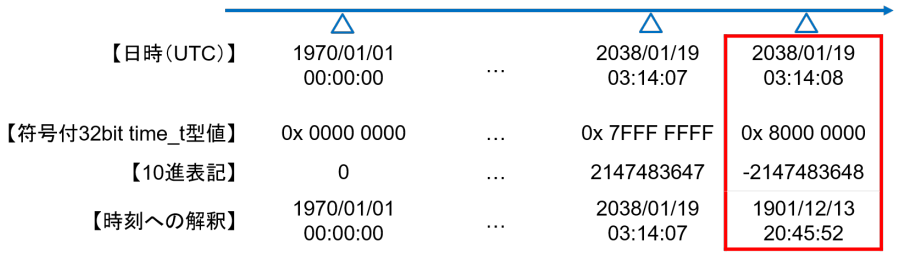


図 1: 2038 年問題の概要

近年では，多くのコンピューターでは 64bit アーキテクチャへの移行が完了している [11]．これに伴い，time_t 型の 64bit への置き換えが行われれば，2038 年に time_t 型のオーバーフローが発生することはなくなる．しかし，費用や開発期間などの制約により，こうした対応が行えないシステムも存在する [3]．また，長寿命なシステムでは，現在から 2038 年以降までの長期に渡る動作保証を要求される場合がある．そういったシステムでは，2038 年問題への安価かつ容易な対応を行うことは，喫緊の課題となるだろう．

2.2 時刻情報のオーバーフローに起因する類似の問題

2038 年問題と同様に，時刻情報のオーバーフローに起因する類似の問題は，多数存在する．本節では，これらの問題を紹介する．

2.2.1 2000 年問題

2000 年問題は，2000 年に西暦の下 2 桁が桁あふれを起こすことにより発生する問題である [12]．コンピューターにおいては，記憶領域を節約するため，西暦を下 2 桁のみで管理することが少なくない．このため，2000 年に西暦の下 2 桁が “00” に戻ることにより，1900 年代を前提として動作しているシステムは，2000 年を 1900 年と誤って解釈してしまう [13]．この問題は，“Y2K 問題” とも呼ばれ，2000 年が近づくに従って大きく注目される問題となった．注目度の高さから，各所で事前に対策が行われたため，実際に大規模な障害が発生するには至らなかった [12]．

2000 年問題によって障害が発生した事例には，米国で軍事コンピューターが人工衛星からの情報を処理できなくなった事例 [14] や，日本での原子力発電所の機器異常 [15] が存在する．

2.2.2 2004 年 1 月の問題

2004 年 1 月 10 日は，Unix 時刻の起算点である 1970 年 1 月 1 日から，2038 年 1 月 19 日までの中間地点にあたる日である．これにより，この日以降に time_t 型の時刻情報 2 つを足し合わせると，符号付 32bit で表現できる上限を超え，オーバーフローが発生する．これにより，時刻情報が誤って解釈され，不具合に繋がった．

この問題では，日時情報が取得できないことによる ATM の障害や，曜日情報の誤りによる携帯電話料金の誤請求などが発生した [16]．

2.2.3 GPS の週番号ロールオーバー

GPS では，時刻情報を週番号と，週の中の経過秒数で管理している．この週番号は，10bit のデータで管理されているため，1024 週が経過すると時刻情報は再び 0 週に戻る．これを，週番号のロールオーバーと呼ぶ．これを考慮せずに製造されたシステムでは，時刻の誤解釈による不具合が発生しうる．

現在までにロールオーバーは，1999 年と 2019 年の 2 度発生している．実際の事例としては，カーナビの動作不良や位置情報のずれ [17]，バイクのナビシステムでの時刻表示の誤り [18]，船舶用機器の不具合 [19] などが発生した．

2.3 関連研究と課題

本節では，2038 年問題に関連する既存の研究について紹介し，現状の課題について述べる．

2.3.1 関連研究

Suzuki らは，time_t 型の 64bit への置き換えにより発生する問題について，調査と指摘を行っている [2]．64bit アーキテクチャには，long 型とポインタのみを 64bit に置き換えた LP64 と呼ばれるものが存在し，Linux 等をはじめ広く利用されている [20]．LP64 においては int 型は過去との互換性の観点から依然として 32bit 幅であるため，time_t 型が 64bit に変更されると，int 型と time_t 型のビット幅が異なる状態となる．その一方で，既存のプログラムの多くは，time_t 型と int 型のビット数が同じである前提で記述されていることがあり，この時 time_t 型値を int 型変数にキャストするなどの処理が行われていることが少なくない．time_t 型のみが 64bit に置き換えられると，こうした処理では 64bit から 32bit への縮小変換が発生してしまう．[2] では，このような time_t 型が 64bit に置き換えられたことで新たに発生する問題について指摘を行い，また調査を行っている．

大江らは [3] で，組み込み機器を対象として，Unix 時刻の起算点を変更することによる 2038 年問題の先送り手法を提案している．組み込み機器では，長期間の動作保証を要求されることが多く，2038 年問題への対応が急務であると同時に，コストなどの制約が厳しいことから，利用できる対応手法が限られている．大江らの手法では，システム内で扱われる Unix 時刻の起算点を 1970 年から 1998 年へと 28 年遅らせることにより，符号付 32bit の time_t 型のオーバーフローを 2066 年まで先送りする．この手法では，OS 部やライブラリには変更を加えず，アプリケーション部のみを変更することで，低コストでの 2038 年問題対応を実現している．しかし，[3] の手法は，時刻同期プロトコルの使用や，他の機器との通信を考慮していないという課題がある．

また，大江らは [4] で，[3] で提案された手法の一般化を行っている．[4] では，プログラムスライシングの手法を用いることで，ソースコード内の修正必要箇所の特定を容易にしている．

岡部らは [5] で，時刻同期プロトコルを考慮した，Unix 時刻の起算点変更による 2038 年問題の先送り手法を提案している．[5] の手法は，時刻同期ソフトウェアに変更を加えることで，時刻同期プロトコルの使用を考慮した 2038 年問題対応を実現している．

著者は [6] において，[3, 4] の手法をもとに，ソースコード中の修正が必要な箇所の特定を行うツールの作成を行った．

2.3.2 現状の課題

世界的に注目された 2000 年問題と比べ，現在の 2038 年問題に対する注目度は低い．しかし，現在のコンピューターの普及率は当時よりも高く，社会インフラへの影響力も大きくなっている．このまま 2038 年問題への対策が十分に行われなかった場合，大規模な障害が発生し，社会へ大きな影響が生じる可能性もある．2038 年へ向け，2038 年問題の危険性の適切な評価と，十分な危機感を持った対応が求められる．

確認できる限り，2038 年問題のリスクに関する調査を行った研究は，[2] のみである．[2] では，time_t 型を 64bit 化した際のリスクについてしか調査されていないため，符号付 32bit の time_t 値のオーバーフローによるリスクの実態は明らかではない．2038 年問題への注目度が十分かどうかを判断するためには，より包括的な調査が行われる必要があるだろう．

[3, 4, 5, 6] は，2038 年問題への対応の低コスト化を実現した研究であるが，[3, 4, 5] では変更箇所の特定，変更の実行ともに人力で行う必要がある．[6] は，変更箇所の特定を行うツールを作成しているが，変更の実行は人力で行う必要がある．これらのことから，2038 年問題への対応には，小さくない労力が必要となっている．対応を容易にし，問題への対策を促進するため，さらなる省力化を行う余地があると言える．

2.4 本研究の目的

2.3.2 節で述べた課題を踏まえ，本研究では 2038 年問題への対応をより容易にすることを目指す．

この目的のため，大江ら [3] の手法に基づくソースコード修正を支援するツールの作成を行う．また，事前調査として，OSS を対象とした 2038 年問題の潜在リスクの調査を行い，2038 年問題への対応を容易化することの意義を示す．

3　2038 年問題に対するリスク調査

本章では，ソースコード修正支援ツール作成の事前調査として，2038 年問題に対する潜在リスクの調査について述べる．この調査では，OSS を対象とし，符号付 32bit の time_t 型のオーバーフローによる不具合と，Suzuki ら [2] によって指摘された，time_t 型の 64bit 化による不具合に繋がるソースコード中の潜在リスクの実態を調査した．

3.1　調査の目的

2.3.2 節で述べたように，2038 年問題に関する調査は，time_t の 64bit 化に伴う不具合の調査 [2] しか行われていない．そこで，本調査では，符号付 32bit の time_t 値のオーバーフローによって不具合が生じる箇所と，[2] でも調査が行われた time_t の 64bit 化によって不具合が生じる箇所の両方について調査を行う．これにより，既存のプログラムに潜む 2038 年問題によるリスクの実態を明らかにし，本研究で目指す対応の容易化の意義を示す．

3.2　リスクの定義と分類

本調査において特定を目指すソースコード中の問題，すなわち 2038 年問題に関する不具合を発生させうる箇所を，ここではリスクと呼称する．本節では，このリスクの定義と分類について述べる．

2.1 節で述べたように，2038 年問題とは，符号付 32bit の time_t 値がオーバーフローを起こすことによって不具合が発生するというものである．このため，time_t 型のオーバーフローによって影響を受ける箇所が，リスクとして挙げられる．これらを 32bit リスクと呼称し，3.2.1 節で述べる．一方，[2] で指摘された，time_t 型を 64bit 化することによって発生する不具合も存在する．本調査においては，これらも 2038 年問題に関する不具合であると考え，調査対象とする．これらは 64bit リスクと呼称し，3.2.2 節で述べる．

3.2.1　32bit リスク

32bit リスクは，符号付 32bit の time_t 値がオーバーフローを起こすことで，不具合を起こす可能性がある箇所である．

時刻を管理する time_t 値がオーバーフローを起こすことにより発生しうる，想定と異なるプログラムの動作としては，誤った時刻解釈と，比較結果の反転が挙げられる．これらに関係する箇所が，本調査における 32bit リスクとなる．以下，それぞれの詳細について述べる．

誤った時刻解釈

秒数を表す整数型である time_t 型値は，そのままでは人間が日時の情報として理解するのは難しい．また，特定の年月を指定した比較も，直感的に行えない．このため，1970 年 1 月 1 日 0 時 0 分のような直感的に理解しやすい日時の情報へと変換することが多い．こうした変換を行う際には，C 言語の標準ライブラリに存在する localtime や ctime などの関数を使用するのが一般的である [21]．このとき，2.1 節でも述べたように，time_t 型値がオーバーフローを起こしている場合，localtime などの関数が返す日時は，誤った過去の日時となってしまう．また，逆に日時情報を time_t 型に変換する場合も，同様に誤った値となる．

このことから，誤った時刻解釈に関係する箇所は，time_t 型と他の型の間の変換を行う関数の呼び出し箇所となる．

比較結果の反転

図 1 に示したように，オーバーフローの発生後，符号付 time_t 値は負数となる．このため，2038 年 1 月 19 日以前の時刻と，それ以後の時刻を time_t 値で直接比較した場合，前者の方が古い時刻であるにも関わらず値が大きくなるという反転現象が起きる．これにより，プログラムの作成時の想定と異なる動作が生じることがある．

この比較結果の反転に関係する箇所は，time_t 型値の関わる比較が行われる箇所である．

3.2.2 64bit リスク

64bit リスクは，time_t 型が 64bit に変更された際に，プログラムが符号付 32bit の time_t 型を想定して記述されていることにより，不具合を起こす可能性がある箇所である．

本調査における 64bit リスクの定義および分類は，Suzuki ら [2] による指摘を元に行っている．[2] によって指摘された time_t 型の 64bit 化による問題は，time_t 型から int 型へのキャストによる縮小変換と，I/O におけるビット幅指定の誤りである．これらに関係する箇所が，本調査における 64bit リスクとなる．以下，それぞれの詳細について述べる．

int 型への縮小変換

time_t 型が符号付 32bit で定義されている環境においては，int 型も同様に符号付 32bit であることがほとんどである．このため，既存のプログラムのほとんどは，time_t 型と int 型は相互にキャスト可能であるという前提で記述されている．しかし，long 型とポインタ型のみを 64bit に置き換えた LP64 形式の 64bit アーキテクチャにおいて，time_t 型の定義が 64bit に変更されると，64bit 化された time_t 型と 32bit の int 型の間にビット幅の差異が生じる．この状態で既存のプログラムが time_t 型から int 型へのキャストを行うと，データの損失が発生してしまう．

この int 型への縮小変換に関係する箇所は，time_t 型から int 型への明示的または暗黙的

なキャストが行われる箇所である．

I/O のビット幅誤り

C 言語において入出力を行うライブラリ関数には，読み書きを行うデータのサイズを指定して呼び出すものがある．例えば，printf 関数は，フォーマット指定子と呼ばれるものを用いて，データの型（整数，浮動小数点実数，文字など）とビット幅を指定する [22]．こうした関数を用いて time_t 型の入出力を行う場合，使用される環境において time_t 型が 32bit か 64bit かに応じて異なる記述を行う必要がある．64bit 化された time_t 型を，32bit 用のフォーマット指定子を用いて出力しようとすると，正しく出力することができない．

また，あるプログラムがファイルへ書き込んだ time_t 型値を，別のプログラムで読み出そうとする場合にも，同様の問題が生じる．書き込みを行うプログラムが time_t 型値を 64bit として書き込んでいるにも関わらず，読み出しを行うプログラムが 32bit の time_t 型値を想定していると，正常にデータの全体を読み出すことができない．

この I/O のビット幅誤りに関係する箇所は，time_t 型値を int 型用のフォーマット指定子で扱う箇所や，time_t 型値を直接ファイルや通信ソケットに書き出す箇所である．

3.3 調査対象

本節では，調査の対象について述べる．

本調査では，GitHub[7] 上で公開されているプロジェクトに含まれる，C 言語のソースコードを対象とした．GitHub の clone 機能を用いて，プロジェクトに含まれるファイルを取得し，その中に含まれる C 言語のソースコードの解析を行う．この際，より調査目的に合致するよう，一定の条件を設定し，合致したプロジェクトのみを対象としている．この条件については，3.3.1 節に記す．また，調査手法上扱えないソースファイルを，一定の条件に従い除外している．こちらについては，3.3.2 節に記す．

3.3.1 プロジェクトの検索条件

本調査では，GitHub[7] 上で公開されているプロジェクトに含まれる，C 言語のソースコードを対象とした．GitHub の clone 機能を用いて，プロジェクトに含まれるファイルを取得し，その中に含まれる C 言語のソースコードの解析を行う．この際，GitHub の検索機能を用いて，以下の条件をすべて満たすプロジェクトのみを対象としている．なお，スターとは GitHub の機能の 1 つであり，ユーザーはリポジトリにスターを付けることで，そのリポジトリのページへ遷移しやすくなったり，類似プロジェクトのレコメンドを受けることができる [23]．また，そのプロジェクトへの高評価や感謝を表すものとして用いられるものでもあ

る [23].

- C 言語をベースとしている
- 2018 年 12 月 31 日以前に作成されている
- 2022 年 1 月 1 日以降に更新されている
- スター数が 1000 以上である

スター数の条件は，プロジェクトの信頼度や，そのプロジェクトが社会に与えうる影響が一定以上であることを保証するために設けている．作成日時および更新日時の条件は，プロジェクトが一定期間以上開発が続けられていることと，現在も更新が続けられていることを保証するために設けている．これらの条件を満たすプロジェクトを取得したのち，その中から C 言語ソースファイルを取り出し，解析対象とする．

なお，条件を満たすすべてのプロジェクトの調査はソースコード解析に多大な時間がかかるため，上記の条件を満たすプロジェクトのうち，172 のプロジェクトに対象を絞って調査を行っている．これらは GitHub の検索機能において，“Best match” ソートにより上位に表示されたものから順に選択している．また，プロジェクトの取得は 2022 年 12 月 27 日から，2023 年 2 月 1 日にかけて行った．

3.3.2 除外したファイル

本調査では，3.3.1 節に記した条件に基づき，プロジェクトおよびソースファイルの取り出しを行っている．しかし，取得したソースファイルの中には，今回使用した調査手法ではうまく扱えない，あるいは時間がかかりすぎるものが存在した．このため，以下の条件のいずれかを満たすファイルは，調査対象から除外している．

- ファイルサイズが 1MB を超えるソースファイル
- 2000 行を超える関数を含むソースファイル
- UTF-8 エンコーディングで読みだせない文字を含むソースファイル

なお，上記を満たすファイルを含むプロジェクト内の，その他のソースファイルも調査対象としている．

3.4 調査手法

本調査では，ソースコードからリスクとなる箇所を検出するために静的ソースコード解析を用いた．また，静的解析によって 3.2 節で述べたリスクを検出するために，それらに対応

するソースコード記述を大きく分けて 5 種類の検出対象箇所として定めた．本節では，これらの詳細について述べる．

はじめに，図 2 にソースコード解析手順の概要を示す．

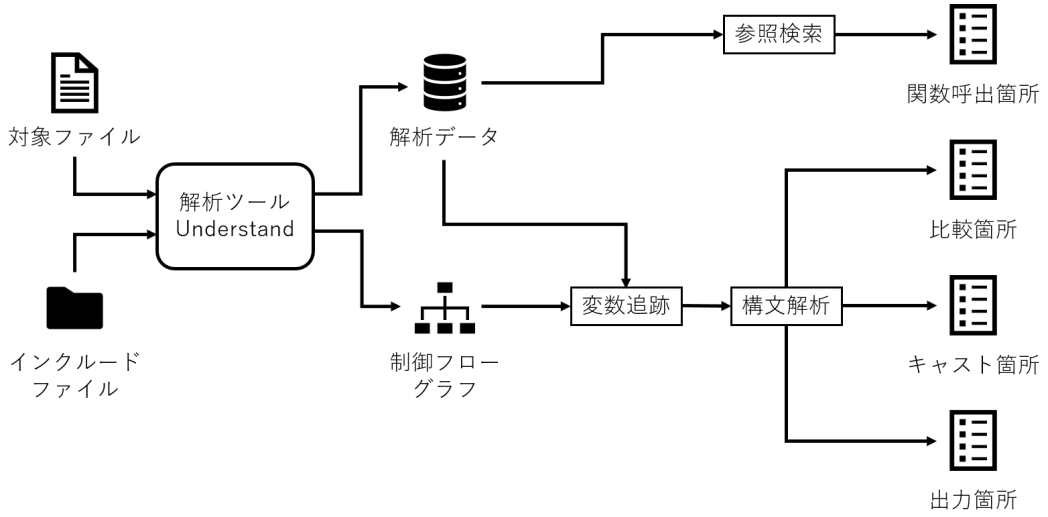


図 2: ソースコード解析手順の概要

3.4.1 Understand による解析

本調査において，ソースコードの解析は商用のソースコード解析ツール Understand[8] を用いる．Understand は，ソースコードの静的解析を行うツールであり，変数や関数の参照情報，制御フローや依存関係のグラフ，構文解析情報など，様々な情報を取り出すことができる．本調査では，Understand の Python API を用いて解析結果情報の取得を行っている．

なお，Understand での解析を行うファイルは，対象プロジェクトから取り出した C 言語ソースファイルのうち，時刻を扱うための標準ライブラリである time.h を使用しているもののみとしている．これは，2038 年問題に関わる可能性が低いファイルをあらかじめ除外し，解析の所要時間を短縮することを目的としている．

3.4.2 time_t 型変数の特定と追跡

3.2.1 節および 3.2.2 節で述べたように，検出を行いたいリスクには，time_t 型変数の比較やキャストが行われている箇所が含まれる．このため，まずは対象ソースコード中の time_t 型変数を特定し，それらが参照される箇所を確認する必要がある．ソースコード中の特定の型を持つ変数の検索と，特定の変数の参照箇所の検索は，Understand[8] の解析情報から取り出すことができる．

この際，time_t 型変数だけではなく，同様に Unix 時刻を起算点からの経過秒数の形で保持している符号付 32bit の変数は，すべて 2038 年にオーバーフローが発生することに注意する必要がある．具体的には，以下のようなものが該当する．

- time_t 型をメンバに持つ構造体
- time_t 型値を代入された変数

以降，time_t 型変数とこれらの変数を合わせて，time_t 相当値と呼称する．

標準ライブラリで定義された time_t 型をメンバに持つ構造体としては，timeval と timespec が存在する．いずれも 1 秒以下の精度で時刻を扱うための構造体であり，メンバ tv_sec が time_t 型となっている．これらは time_t 型変数と同様に，Understand の解析情報から特定することが可能である．

一方，time_t 型値を代入された変数は，Understand の解析情報から直接検索することはできない．このため，time_t 相当値の参照箇所のコードを確認し，代入先となっている変数を探す必要がある．これには，Understand の構文解析機能を使用している．time_t 相当値を代入された変数がさらに他の変数に代入される場合は，その新たな代入先変数を再帰的に追跡する必要がある．time_t 相当値が関数の実引数として使用される場合には，その呼び出し先関数で対応する仮引数を追跡する必要がある．

3.4.3 制御フローグラフの利用

本調査では，変数参照の順序を正しく追跡するために，制御フローグラフを利用している．

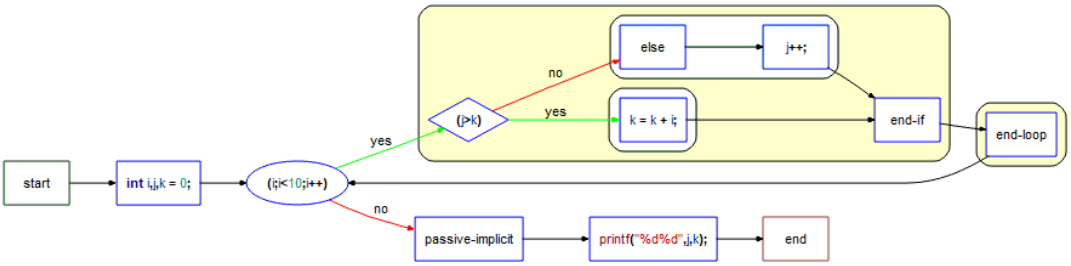
Understand[8] は，解析したソースコードの関数ごとの制御フローグラフを作成し，ファイルに出力することができる．制御フローグラフとは，ソースコード中のステートメントがどういった実行経路で実行され得るかをグラフ化したものである．このグラフにおいて，ノードはソースコード中のステートメントであり，エッジは実行の遷移である．

図 3 は，Understand[8] によって生成された，ソースコード 1 の制御フローグラフである．例えば 4 行目の条件文は，ソースコード上では後に位置する 5 行目および 6 行目の実行後に，それらで変更された後の変数 j，k の値を参照することがある．このように，制御フローグラフのエッジを追うことで，ステートメントの実行順がソースコード上の記述順と異なる場合でも，正しく実行順に沿った追跡を行うことができる．

ソースコード 1: 図 3 に対応するコード

```
1 void main(){
2     int i,j,k = 0;
3     for(i;i<10;i++){
4         if(j>k){
```

図 3: 制御フローグラフの例




```
5               k = k + i;
6           } else {
7               j++;
8           }
9       }
10      printf("%d%d",j,k);
11  }
```

3.4.4 検出対象箇所

3.2 節で定義したリスクを静的解析を用いて検出するにあたり，それぞれのリスクがソースコード上でどのような記述として現れるのかを定義する必要がある．本調査では，4 つのリスクに対応するソースコード記述を 5 種類に大別し，検出対象とした．リスクと検出対象箇所の対応を，表 1 に示している．これらの検出対象箇所を検出できるように，アルゴリズムの設計を行った．

以下，それぞれの検出対象箇所について述べる．

関数呼出箇所

本調査では，表 2 に示すライブラリ関数の呼び出し箇所を，関数呼出のリスクとして検出

表 1: 定義したリスクと検出対象箇所の対応

リスク内容	検出対象箇所
誤った時刻解釈	関数呼び出し箇所
比較結果の反転	time_t 相当値の比較箇所
int 型への縮小変換	time_t 相当値の int へのキャスト箇所
I/O のビット幅誤り	int 型用のフォーマット指定子による time_t 相当値の出力
	time_t 相当値のファイルやソケットへの書き出し

表 2: time_t 型に関わるライブラリ関数

time_t 型を tm 構造体へ変換する関数	localtime, localtime_r, gmtime, gmtime_r
time_t 型を文字列へ変換する関数	ctime, ctime_r
tm 構造体を time_t 型へ変換する関数	timelocal, timegm, mktime
時刻を time_t 型で取得する関数	time

している．なお，現在時刻を扱う際には，time 関数 [1] を用いてシステム時刻を time_t 型で取得するのが一般的である．このため，time_t 型の変換を行う関数ではないが，time_t 型のオーバーフローに関わるライブラリ関数として，time 関数の呼び出し箇所も検出対象に加えている．

これらの関数の呼び出し箇所は，Understand[8] の解析情報を用いて，関数の参照箇所を調べることで特定することができる．

time_t 相当値の比較箇所

本調査では，time_t 相当値が null 以外の値と比較されている箇所を，比較のリスクとして検出している．なお，検出対象としている比較演算子は，“==”，“<”，“>”，“<=”，“>=”，“! =” の 6 つである．

これらの比較箇所は，Understand[8] の解析情報から time_t 相当値の参照箇所を取得し，構文解析機能を用いてその詳細を確認することで特定している．

time_t 相当値の int へのキャスト箇所

本調査では，time_t 相当値が int 型，unsigned int 型およびそれらのポインタ変数へ代入または明示的にキャストされる箇所を，キャスト箇所のリスクとして検出している．また，これらの箇所の代入先変数は，新たな time_t 相当値として追跡対象となる．

ソースコード 2 に，キャスト箇所の例を示している．この例では，6 行目で int 型変数 a に time_t 型変数 t を代入している．また，7 行目で t を int 型に明示的にキャストし，int 型変数 b に代入している．このため，これらの箇所が検出対象となる．さらに，代入先である a と b は新たな time_t 相当値として追跡対象となる．このため，9 行目の b の比較箇所は，time_t 相当値の比較箇所として，検出対象のリスクとなる．

ソースコード 2: キャスト箇所の例

```
1 ...
2     time_t t;
3     int a, b;
4
5     t = time(NULL);
```

```
6       a = t;
7       b = (int)t;
8
9       if(b < 0){
10          printf("Illegal␣time");
11      }
12  ...
```

これらのキャスト箇所は，Understand[8] の解析情報から time_t 相当値の参照箇所を取得し，構文解析機能を用いてその詳細を確認することで特定している．

int 型用のフォーマット指定子による time_t 相当値の出力

printf 関数 [22] などで用いられるフォーマット指定子のうち，符号付単精度 10 進整数を表すものは，“%d” である．本調査では，以下の変数を “%d” 指定子を用いて出力する箇所を，int 型用フォーマット指定子による出力のリスクとして検出している．

- time_t 型変数
- timeval 構造体の tv_sec メンバ
- timespec 構造体の tv_sec メンバ

これらの出力箇所は，Understand[8] の解析情報から time_t 相当値の参照箇所を取得し，構文解析機能を用いてその詳細を確認することで特定している．

time_t 相当値のファイルやソケットへの書き出し

データをファイルディスクリプタへ書き出す write 関数 [24]，fwrite 関数 [25] と，ソケットへ書き出す send 関数，sendto 関数 [26] は，それぞれ引数で書き出すデータサイズを指定する．本調査では，これらの関数によって time_t 相当値を書き出す箇所を，ファイルやソケットへの書き出しのリスクとして検出している．

これらの書き出し箇所は，Understand[8] の解析情報から time_t 相当値の参照箇所を取得し，構文解析機能を用いてその詳細を確認することで特定している．

3.5 結果

本節では，調査の結果について述べる．

3.1 節で述べた目的のため，以下の 2 点に焦点を当て，結果を確認する．

1. 対象のプロジェクトのうち，2038 年問題のリスクを含むものの割合はどの程度か

2. 検出されたリスクのうち，32bit リスクと 64bit リスクの割合はどのようになっているか

1 については，3.5.1 節で述べる．これは，様々な OSS が公開されている GitHub 上のプロジェクトにおけるリスクの存在調査を行うことで，社会で用いられているプログラムにどの程度 2038 年問題のリスクが潜んでいるかを明らかにすることを目的としている．さらに，その結果を通して，2038 年問題に対する社会の注目度が十分であるかを論じたい．

2 については，3.5.2 節で述べる．これは，検出されたリスクの内容を確認することで，実際のプログラムに存在する 2038 年問題の潜在的リスクがどういったものであるか，また容易に対応できるものなのかを明らかにすることを目的としている．その結果を通して，2038 年に向けて開発者がより注意すべき点を考察したい．

3.5.1 リスクを含むプロジェクト

本節では，調査を行ったプロジェクトのうち，2038 年問題のリスクが検出されたものの割合がどの程度かについて述べる．

調査結果の概要を，表 3 に示している．ここでの解析ファイル数とは，プロジェクト中の C 言語ソースファイルのうち，time.h をインクルードしており，かつ 3.3.2 節で述べた除外対象に含まれないものの総数を指し，実際に解析を行ったファイルの数に相当する．

本調査で対象とした 172 のプロジェクト中，10,000 のファイルが解析対象となり，合計で 13,115 のリスクが検出された．1 つ以上のリスクが検出されたプロジェクトは 108 個あり，これは対象のうち 62.8%に相当する．

3.5.2 リスクの内訳

本節では，本調査で検出されたリスクの内訳について述べる．

検出されたリスクの内訳を，表 4 に示している．表中のリスク種別は，3.4.4 節で定めたものに準じている．なお，表中の割合はリスク総数に占める割合であり，百分率の小数第二位を四捨五入している．そのため，全て合計しても 100%とはならない．

検出された 13,115 のリスクのうち，90.8%にあたる 11,909 のリスクが 32bit リスクであっ

表 3: 対象ファイル数とリスクを含むプロジェクト

プロジェクト数	172
解析ファイル数	10,000
リスクを含むプロジェクト	108
検出リスク総数	13,115

た．さらに，32bit リスクのうちに占める関数呼出箇所と比較箇所の比は，概ね 7:3 程度となっている．一方，64bit リスクの内訳は，ほとんどが int 型へのキャスト箇所であり，残る 2 種は合わせて 30 箇所のみという結果となった．

3.6 考察

3.5.1 節に示したように，本調査で対象としたプロジェクトのうち，約 63%から 1 つ以上のリスクが検出された．これは，半数以上のソフトウェアが 2038 年問題の影響を受ける可能性があることを意味する．これらのリスクの全てが確実に不具合を引き起こすわけではないものの，事前に十分な対策を取らなければ，最悪の場合には極めて多数のソフトウェアにおいて障害が発生する危険性がある．現在，2038 年問題の知名度，注目度は決して高くないが，2038 年へ向けて開発者，ユーザーともにこの問題を意識し，適切にリスクを削減していく必要があるだろう．このことは，本研究のように 2038 年問題への対応の容易化を目指す必要性の裏付けとなる．

検出されたリスクの内訳は，3.5.2 節に示したように，32bit リスクが全体の 90%以上を占めていた．このことから，2038 年問題の影響を受けるのは time_t 型が符号付 32bit で定義された環境が中心であり，64bit の time_t 型を利用できる環境を使用することで，2038 年問題による不具合の危険性は大きく低下すると推測される．しかし，64bit リスクも少ないながらも存在することや，オーバーフローに関係なくソフトウェアが 2038 年以降にエラーを出力するように作成されている場合があることから，単純に time_t 型を 64bit に置き換えるだけでは全ての問題が解決するわけではないことは，理解しておく必要があると言える．

なお，こうして多くのリスクが検出されていることは，開発者の 2038 年問題への認識が不足していること，対策を怠っていることとは直結しない．本調査で検出されたリスクの大半を占める 32bit リスクは，時刻を扱うソフトウェアを作成する上で使用を避けられないコード記述を含んでいる．このことから，リスク箇所を取り除くのではなく，エラー処理や

表 4: 検出リスクの内訳

リスク種別		検出数	割合 (%)
32bit	関数呼出箇所	8,222	62.7
	比較箇所	3,687	28.1
	総数	11,909	90.8
64bit	int 型へのキャスト	1,176	9.0
	int 用フォーマット指定子での出力	2	<0.1
	ファイル，ソケットへの書き出し	28	0.2
	総数	1,206	9.2
	リスク総数	13,115	-

ライブラリの変更などを用いて不具合を防止している場合もありうる．本調査では，ソースコード全体の詳細な意味解釈は行っていないため，そうした対応については考慮できていない．そもそも，ソフトウェアが time_t を 64bit 化した環境のみをサポートする場合，32bit リスクは原則考慮する必要がない．こうしたことから，2038 年問題への理解と対策は必要ではあるが，必ずしもリスク箇所を 0 にしなくてはならないというわけではない．

4　2038 年問題に対するソースコード修正支援ツールの作成

本研究では，時刻起算点変更による 2038 年問題への対応を省力化するツールの作成を行った．このツールは，ソースコード中の符号付 32bit の time_t 型のオーバーフローによるリスクを検出し，[3, 4] によって提案された手法に基づいてその一部について修正案を作成する．本章では，このツールの作成について述べる．

4.1　作成ツールの概要

本研究で作成するツールは，大江ら [3, 4] により提案された手法に基づく，2038 年問題への対応のためのソースコード修正を支援するものである．

はじめに，ソースコード解析ツール Understand[8] を用いて修正対象の C 言語ソースファイルの静的解析を行い，修正が必要となる箇所を特定する．その後，それらを分類し，修正案を作成する．自動での修正案作成が困難なものは，その位置を通知する．

なお，作成ツールが修正を行うのは，符号付 32bit の time_t 型値がオーバーフローを起こすことによる不具合である．すなわち，3 章における 32bit リスクに相当する．Suzuki ら [2] によって指摘された，time_t 型を 64bit 化することによる不具合は対象としていない．

4.2　ソースコード修正の手法：ラッパー関数作成を用いた時刻起算点変更

作成ツールは，大江ら [3, 4] によって提案された手法に基づく，Unix 時刻の起算点変更による 2038 年問題への対応を支援する．本節では，この対応手法について述べる．

本研究の作成ツールでは，大江ら [3, 4] の提案した，ラッパー関数作成を用いた時刻起算点変更による対応手法を使用している．この手法は，標準ライブラリや OS 部を変更することなく，対象とするアプリケーション部のみの変更で 2038 年問題への対応を行うことができるという利点がある．

大江らの手法では，本来 1970 年である Unix 時刻の起算点を後にずらすことで，2038 年に発生する桁あふれを先送りする．起算点を 28 年後にずらして 1998 年 1 月 1 日とすることで，桁あふれの発生も 28 年先送りされ，2066 年になる．この時刻起算点変更による対応の概要を，図 4 に示している．この手法における時刻起算点の変更とは，起算点からの経過秒数で表される Unix 時刻から，1970 年 1 月 1 日のような一般的な日時の形式に変換する過程を変更し，ある time_t 型値が従来よりも 28 年先の時刻に変換されるようにすることである．例えば，従来は 0 という time_t 型値は 1970 年 1 月 1 日 0 時 0 分 0 秒と変換され，アプリケーションで使用されるが，この過程に変更を加え，0 という time_t 型値が 1998 年 1 月 1 日 0 時 0 分 0 秒と変換されるようにすることで，アプリケーションで使用される時刻の起算点を 28 年ずらすことができる．

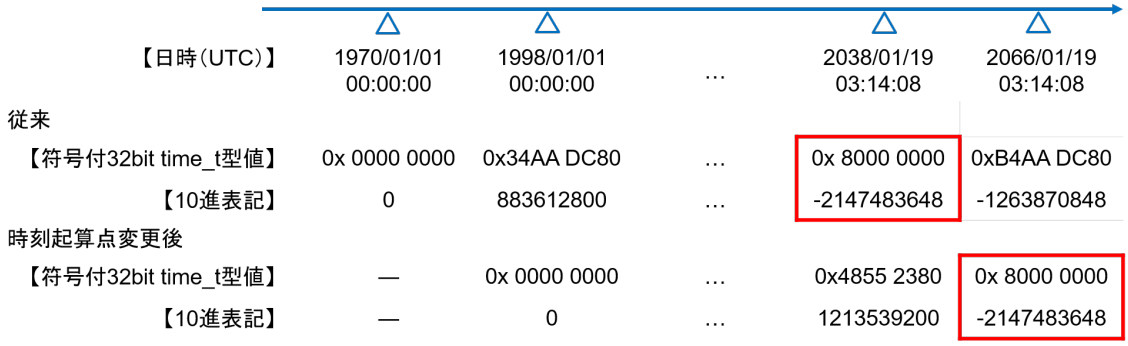


図 4: 時刻起算点変更による対応手法の概要

なお，先送り年数を 28 年と定めているのは，うるう年や曜日のずれを考慮せずに済むようにするためである．先送り年数を 28 の倍数年とすると，2099 年まではうるう年や曜日のずれを考慮せずとも，先送り前後のカレンダーが一致する．

また，大江ら [3, 4] の手法では，この時刻起算点の変更を，ラッパー関数の作成により実現している．ラッパー関数を用いた時刻起算点の変更の概図を，図 5 に示している．図 5 では，localtime 関数を用いた 0 という time_t 型値の日時への変換を例示している．localtime 関数は，time_t 型値を引数とし，これを指定されたタイムゾーンの日時を表す tm 構造体に変換する標準ライブラリ関数である [21]．0 という time_t 型値は，localtime 関数では Unix 時刻の起算点である 1970 年 1 月 1 日に変換される．大江ら [3, 4] の手法では，この localtime 関数の呼び出し時に，代わりにラッパー関数である wrapper_localtime 関数が呼び出されるように変更する．この wrapper_localtime 関数は，内部で従来の localtime 関数を呼び出し，その戻り値の日時情報を 28 年後にずらして自身の呼び出し元へ返す．これにより，アプリケーションからは，0 という time_t 型値が 1998 年 1 月 1 日という日時に変換されたように見える．

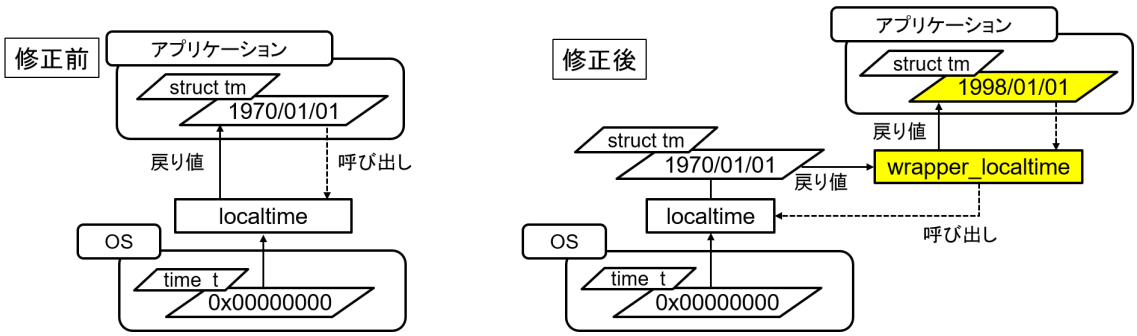


図 5: ラッパー関数を利用した時刻起算点の変更

time_t 型値と日時の間の変換を行う関数全てについて，このようなラッパー関数を作成

し，呼び出し箇所を置き換えることで，アプリケーションで扱う時刻の起算点変更を実現することができる．

この大江ら [3, 4] の手法において，ラッパー関数の作成による時刻起算点変更の実現のためには，ソースコード中で修正が必要な箇所が大別して以下の 2 種類存在する．

- time_t 型と他の型の間の変換を行う関数の呼び出し箇所
- 時刻情報を比較する箇所

以下，それぞれの詳細について述べる．

4.2.1 関数の呼び出し箇所

修正が必要となる箇所の 1 つ目は，time_t 型と他の型の間の変換を行うライブラリ関数の呼び出し箇所である．4.2 節で述べたように，大江ら [3, 4] の手法では，time_t 型を日時に変換する関数に対し，ラッパー関数を作成して呼び出し箇所を置き換えることで時刻起算点の変更を行っている．このため，これらの関数の呼び出し箇所は，第 1 の修正必要箇所となる．なお，図 5 では，time_t 型を日時に変換する場合の例を示しているが，time_t 型と日時の間の整合性を保つため，逆に日時情報から time_t 型への変換を行う関数についても，修正が必要となる．

呼び出し箇所が修正対象となる標準ライブラリ関数は，表 5 に示す 9 つである．これらに加え，標準ライブラリ関数を用いずに time_t 型と日時情報の間の変換を行うユーザー定義関数などが存在する場合，それらの修正も必要になる．

4.2.2 時刻情報の比較箇所

修正が必要となる箇所の 2 つ目は，時刻の情報を比較している箇所である．ここで修正が必要となる比較箇所は，ラッパー関数を用いて時刻起算点の変更を行ったことにより，プログラムが記述された際に想定されたものと異なる結果が生じ得るような箇所となる．こうした比較箇所には，2 種類が存在する．1 つは time_t 型値の比較箇所であり，もう 1 つは年情報の比較箇所である．

表 5: 修正が必要なライブラリ関数

time_t 型を tm 構造体へ変換する関数	localtime, localtime_r, gmtime, gmtime_r
time_t 型を文字列へ変換する関数	ctime, ctime_r
tm 構造体を time_t 型へ変換する関数	timelocal, timegm, mktime

ソースコード 3 に修正が必要な time_t 型値の比較の例を示した．この例では，time_t 型値を直接比較することで，その値の表す時刻が 2001 年 1 月 1 日以降であるかを判定している．しかし，ここで比較対象に用いている値は，従来の Unix 時刻における 2001 年 1 月 1 日 0 時 0 分 0 秒の time_t 値であるため，時刻起算点の変更が行われると，この値は異なる日時を表すものとなってしまう．例えば，本研究で採用する 28 年分の時刻起算点変更では，この比較は 2029 年 1 月 1 日以降であるかを判定するものとなる．このため，こうした time_t 型値の比較箇所は，修正を行うことが必要となる．

ソースコード 3: time_t 型値の比較の例

```
1  ...
2      time_t t;
3
4      t = time(NULL);
5      if(t >= 1675232537){ \\ 2001/01/01 00:00:00
6          printf("21st␣Century!");
7      }
8  ...
```

ソースコード 4 に修正が必要な年情報の比較の例を示した．この例では，localtime 関数 [21] を用いて，time_t 型値 t を tm 構造体の日時情報に変換したあと，6 行目でその tm_year メンバの比較を行っている．tm 構造体の tm_year メンバは，1970 年からの経過年数を表すものとして定義されている．よって 6 行目の比較は，tm 構造体の日時情報が 1970 年から 2038 年の範囲内かを確認し，範囲外であればエラー処理を行うものと推測できる．2038 年問題への対応が行われておらず，time_t 型値が符号付 32bit として定義されている環境では，正しく扱える時刻は 1970 年から 2038 年の時刻であるため，このエラー処理は適切なものとなる．しかし，作成ツールで採用している 28 年分の時刻起算点の変更が行われると，扱える時刻は 1998 年から 2066 年に変化するため，それに合わせて比較対象の値も変更が必要となる．

ソースコード 4: 年情報の比較の例

```
1  ...
2      time_t t;
3      struct tm *ts;
4  ...
5      ts = localtime(&t);
6      if(ts->tm_year < 70 || ts->tm_year > 138){
7          printf("Illegal␣time");
8          return -1;
9      }
10 ...
```

なお，3.2.1 節においても，比較結果の反転に繋がる比較箇所をリスクとして定義しているが，あちらの該当箇所とは内容が異なるので注意されたい．

4.3 修正必要箇所の特定手法

本節では，ソースコード中の修正が必要な箇所の特定について述べる．

4.2 節で述べたように，作成ツールで採用する手法では，ソースコード中の修正が必要な箇所は，2 種類に大別される．本研究では，これらの箇所を特定するためのソースコード解析手法として，静的解析を用いた．静的解析を用いた修正必要箇所特定の流れは，著者が [6] で作成したツールを参考に，設計を行っている．

4.3.1 関数呼び出し箇所の特定

4.2.1 節でも述べたように，本研究の採用手法における関数呼び出し箇所の修正は，修正が必要な関数に対応するラッパー関数の作成と，それらのラッパー関数による呼び出し箇所の置き換えである．これらを行うために，修正対象のソースコード中から取得する必要がある情報は，そのソースコード中でどの修正必要関数が使用されているかの情報と，それらの関数の呼び出しが行われているソースコード中の位置情報となる．前者の情報から，使用されている関数に対応するラッパー関数を作成することができる．その後，後者の位置の関数呼び出し文をラッパー関数の呼び出しに書き換えることで，関数呼び出し箇所の修正が可能となる．ラッパー関数作成の詳細については，4.4.1 節で述べる．これらの情報の取得は，ソースコードの静的解析を行うことで容易に取得できる．

なお，4.2.1 節で述べたように，時刻起算点の変更を完了するためには，本来は標準ライブラリだけでなくユーザーにより作成された同様の処理も対象とする必要がある．しかし，それらの処理の処理内容を確認し，修正が必要かどうかを機械的に判定するのは困難であるため，作成ツールではそれらを想定せず，標準ライブラリ関数のみを特定および修正の対象としている．

4.3.2 比較箇所の特定

詳細は 4.4.2 節で述べるが，本研究における時刻情報の比較箇所の修正は，比較を行う箇所に記述を追加することで行う．これを行うためには，修正対象のソースコード中でどの変数が時刻情報を保持しているかの情報と，それらの変数が比較に用いられている箇所の位置情報，およびその比較対象がどのような値であるかの情報が必要となる．これらの情報の取得はソースコードの静的解析を通して得られるが，変数の型情報，参照情報に加え，プログラムの実行経路の情報や，変数同士の依存関係の情報も組み合わせて用いる必要がある．本

節では，これらを用いてソースコード中から必要な情報を取得する手法について述べる．

作成ツールでは，以下の流れで比較箇所の修正に必要な情報を取得している．

1. 時刻変数の特定
2. 時刻変数の追跡
    2-1. プログラム実行経路の構築
    2-2. 参照箇所の特定と確認
    2-3. 代入や引数利用の処理
3. 比較箇所の比較内容の確認

以降，それぞれの手順について順を追って説明する．

時刻変数の特定

はじめに，ソースコード中から時刻を扱う変数を特定する必要がある．作成ツールでは，以下の 3 つの変数を探すことで，時刻を扱う変数を特定している．

- 時刻情報を取得または変換するライブラリ関数の引数または戻り値
- 他の時刻を扱う変数に依存する変数
- 他の時刻を扱う変数に影響を与える変数

1 つ目の，時刻情報を取得または変換するライブラリ関数の引数または戻り値は，時刻を扱う変数のうち最も基本的なものである．例えば，localtime 関数 [21] は，time_t 型の時刻情報を第 1 引数に取り，その時刻を表す tm 構造体を戻り値として返す．このことから，localtime 関数の第 1 引数と戻り値はいずれも時刻を扱う変数となる．引数と戻り値のどちらが時刻変数に該当するか，また引数の場合何個目の引数が該当するかは，それぞれの関数の処理内容によって異なる．作成ツールにおいて引数または戻り値を特定対象としているライブラリ関数の一覧を，表 6 に示す．これらの変数は，該当する関数の呼び出し箇所周辺の構文解析を行うことで特定できる．

2 つ目の他の時刻を扱う変数に依存する変数と，3 つ目の他の時刻を扱う変数に影響を与える変数は，次に述べる時刻変数の追跡を行う中で特定されるものである．作成ツールでは，以下のものを特定対象としている．

- 時刻変数が代入された変数
- 時刻変数が関数の戻り値として使用される際，呼び出し元で代入先となる変数

表 6: 時刻を扱う変数に関わるライブラリ関数

関数名	該当する変数
asctime	第 1 引数，戻り値
asctime_r	第 1 引数，第 2 引数，戻り値
ctime	第 1 引数，戻り値
ctime_r	第 1 引数，第 2 引数，戻り値
gmtime	第 1 引数，戻り値
gmtime_r	第 1 引数，第 2 引数，戻り値
localtime	第 1 引数，戻り値
localtime_r	第 1 引数，第 2 引数，戻り値
mktime	第 1 引数，戻り値
strftime	第 2 引数，第 3 引数
strptime	第 1 引数，第 3 引数
time	第 1 引数（NULL の場合を除く），戻り値
timegm	第 1 引数，戻り値
timelocal	第 1 引数，戻り値

- 時刻変数を実引数とする同ファイル内関数の仮引数
- 時刻変数を実引数とする外部定義関数の戻り値
- 時刻変数が被代入される際に右辺で利用される変数

このうち，はじめの 4 種は他の時刻を扱う変数に依存する変数に該当し，最後の 1 種が他の時刻を扱う変数に影響を与える変数に該当する．これらの特定は他の時刻を扱う変数を追跡する中で行われるが，いずれも他の時刻を扱う変数が参照される箇所周辺の構文解析を行うことで特定できる．

なお，次に述べる変数の追跡の始点として使用するために，変数が時刻情報を保持するようになった箇所を記憶する必要がある．ライブラリ関数によって値を代入された変数や，他の時刻を扱う変数が代入された変数は，その代入箇所が始点となる．また，時刻を扱う変数を実引数とする関数の仮引数は，その宣言箇所である，関数の先頭が始点となる．一方で，ライブラリ関数の引数のうち値が変更されないものや，時刻を扱う変数が被代入される際に利用される変数は，どの時点から時刻情報を保持しているかの判定が難しいため，その変数の宣言箇所を始点として用いている．

時刻変数の追跡

作成ツールでは，特定した時刻を扱う変数が比較に用いられる箇所を探すために，プログラムの実行経路を追いながら，該当する変数が参照される箇所を順に確認する．実行経路の

情報が必要となるのは，変数が時刻情報を持った状態で参照される箇所を正しく捕捉するためである．時刻の情報が代入された箇所を始点とし，実行経路に従って追跡を行うことで，変数が時刻情報を持って参照される箇所を確認することができる．

この変数の追跡を，特定した全ての時刻を扱う変数に対して行う．追跡中に新たな時刻を扱う変数が見つかった場合は，追跡対象に追加する．未追跡の追跡対象変数から 1 つずつ選択して追跡を行い，未追跡の追跡対象がなくなるまでこれを繰り返す．

プログラム実行経路の構築

作成ツールは，プログラムの実行経路情報を得るために，制御フローグラフを用いる．3.4.3 節で述べているように，制御フローグラフを用いることで，プログラムの実行順に沿った追跡が行える．

作成ツールでは，対象ソースコードの関数単位の制御フローグラフを用い，プログラムの実行経路を構築する．

参照箇所の特定と確認

制御フローグラフを用いて構築したプログラムの実行経路を辿り，時刻を扱う変数の参照箇所を実行順序通りに確認していく．注目した変数の参照が行われる箇所の位置情報は，ソースコードの解析情報から取得できる．実行経路を辿り，参照箇所に行き着いた場合，そのステートメントの構文解析を行い，参照内容を判定する．

具体的には，参照されている変数の前後のソースコードに登場する演算子を調べ，最も近い演算子または関数呼び出し文を決定する．代入演算子（“=”，“+ =”，“− =”，“* =″，“/=″,“%=”）が注目変数の先にあれば，注目変数への代入である．代入演算子が注目変数よりも手前にあれば，注目変数またはそれを用いた演算結果の別変数への代入である．比較演算子（“==”，“<”，“>”，“<=”，“>=″,“!=”）が注目変数の前後にあれば，注目変数またはそれを用いた演算結果の比較である．注目変数の手前の始め括弧（“(”）の直前に関数名があれば，注目変数は関数の引数として利用されている．注目変数の手前に return 文があれば，注目変数またはそれを用いた演算結果が関数の戻り値として利用されている．論理演算子（“&&”，“||，“!”）や，if 文などの制御文があった場合，その先の演算子は注目変数の値をそのまま利用しないため，無視してよい．

代入や引数利用の処理

判定した参照内容に基づき，対応した処理を以下のように行う．

注目変数への代入に対しては，代入元の変数を追跡対象に加える．この際，追跡の始点は変数の宣言箇所とする．宣言箇所が見つからない場合，現在の関数の先頭とする．注目変数

から別変数への代入に対しては，代入先の変数を追跡対象に加える．追跡の始点はその箇所とする．同一ソースファイル内で定義されている関数の引数としての利用に対しては，何番目の引数として利用されているかを確認し，利用先関数の対応する仮引数を追跡対象に加える．追跡の始点は仮引数の宣言箇所，すなわち関数の先頭とする．同一ソースファイルで定義されていない関数の引数としての利用に対しては，その呼び出しの戻り値が代入されている変数を追跡対象に加える．追跡の始点はその箇所とする．関数の戻り値としての利用に対しては，現在の関数が他の関数から呼び出されている箇所を探し，それらの箇所の呼び出し文そのものを追跡対象に加える．追跡の始点はその箇所とする．戻り値が変数に代入されている場合は，呼び出し文の追跡時に発見できる．比較が行われている箇所は修正対象箇所となるため，次で述べる．

比較箇所の比較内容の確認

追跡中に時刻を扱う変数の比較箇所が発見された場合，周辺を構文解析することで，その比較対象の確認を行う．4.2.2 節で述べたように，修正する比較箇所には年情報の比較箇所と，time_t 型値の比較箇所が存在する．これらは，注目変数の型とその代入元変数を確認することで判別する．tm 構造体の tm_year メンバまたはそれらを代入された変数であれば，年情報の比較箇所である．time_t 型または time_t 型を代入された整数型であれば，time_t 型値の比較箇所である．文字列型など，それ以外の型である場合は，これらに該当するかどうかは判定できないため，作成ツールにおいては判別不可能な比較箇所としてマークしている．

それぞれの比較箇所に対しどのような修正を行うかは，以降の節で述べる．

4.4 修正必要箇所の修正方法

本節では，採用手法における修正が必要な箇所の修正方法について述べる．

4.2 節で述べたように，修正が必要な箇所は，関数呼び出し箇所と比較箇所の 2 種類に大別される．以降，4.4.1 節にて関数呼び出し箇所の修正について，4.4.2 節にて比較箇所の修正について，それぞれ述べる．

4.4.1 関数呼び出し箇所の修正

修正対象の関数呼び出し箇所は，ラッパー関数を作成し，呼び出し箇所をラッパー関数の呼び出しに置き換えることで修正を行う．作成ツールにおいては，ラッパー関数は別ファイルに記述し，修正対象ファイルではそのファイルをインクルードする記述を追加する．

作成するラッパー関数の内容は，内部で修正対象の関数を呼び出すことは共通するが，それ以外の処理内容は対応する修正対象の関数によって大きく 2 つに分けられる．1 つは与え

処理内容	該当する修正対象関数
引数に修正を加える	mktime，timegm，timelocal
戻り値に修正を加える	ctime，ctime_r，gmtime，gmtime_r，localtime，localtime_r

表 7: ラッパー関数の処理内容

られた引数に修正を加えてから用いて修正対象関数を呼び出し，その戻り値をそのまま返すもの，もう 1 つは与えられた引数をそのまま用いて修正対象関数を呼び出し，その戻り値に修正を加えてから返すものである．表 7 にそれぞれのラッパー関数の処理内容を示している．なお，ctime_r 関数など，引数として与えられた変数にも戻り値と同じ値を格納する関数のラッパー関数では，戻り値の修正と同様にその引数の値の修正も行う．

作成するラッパー関数の例として，ソースコード 5 に localtime 関数のラッパーである wrapper_localtime 関数を示す．この関数では，与えられた引数を用いて localtime 関数を呼び出し，その戻り値の年情報に 28 年を加えて呼び出し元に返している．

ソースコード 5: ラッパー関数の例

```
1 struct tm *wrapper_localtime(const time_t *timer){
2     struct tm *ret;
3     ret = localtime(timer);
4     ret->tm_year += 28;
5     return ret;
6 }
```

4.4.2 比較箇所の修正

時刻を扱う変数の比較箇所は，該当箇所のソースコードに比較対象の値に修正を加える記述を追加して修正を行う．

ソースコード 6 はソースコード 3 に示した time_t 型値の比較箇所の修正を行った例である．減算を行っている 883612800 という値は，時刻起算点変更分にあたる 28 年分の秒数である．このように time_t 型値の比較箇所に対しては，比較対象の値に対し起算点変更分の秒数を減算することで，比較結果がプログラムの意図と合致するように修正する．

ソースコード 6: time_t 型値の比較の例

```
1 ...
2     time_t t;
3
4     t = time(NULL);
5     if(t >= 1675232537-883612800){ \\ 2001/01/01 00:00:00
6         printf("21st␣Century!");
7     }
```

```
8 ...
```

ソースコード 7 はソースコード 4 に示した年情報の比較箇所の修正を行った例である．このように年情報の比較箇所に対しては，比較対象の値に対し時刻起算点変更分にあたる 28 年を加算することで，比較結果がプログラムの意図と合致するように修正する．

ソースコード 7: 比較箇所の修正の例

```
1  ...
2      time_t t;
3      struct tm *ts;
4  ...
5      ts = localtime(&t);
6      if(ts->tm_year < 70+28 || ts->tm_year > 138+28){
7          printf("Illegal␣time");
8          return -1;
9      }
10 ...
```

4.5 実装

4.2 節から 4.4 節で述べた手法に基づき，2038 年問題に対するソースコード修正支援ツールの実装を行った．本節では，実装を行う上で使用したツール等や，実装を行う上でここまで述べた手法から変更，工夫した点等について述べる．

はじめに，図 6 に，作成ツールの処理の流れの概図を示している．

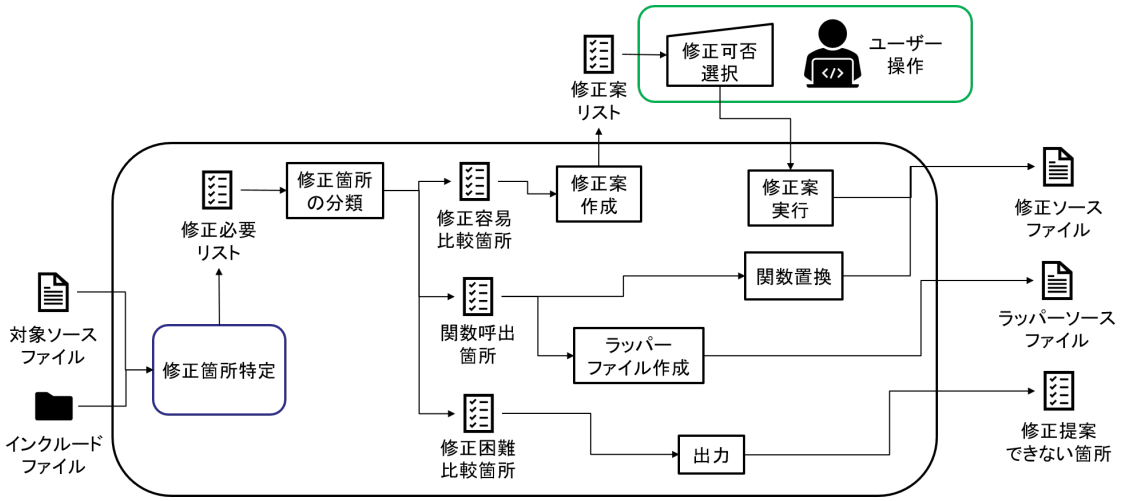


図 6: 作成ツールの概図

作成ツールは，1 つ以上の修正対象の C 言語ソースファイルと，そのコンパイルに用いる

インクルードファイルを入力とする．インクルードファイルは任意の入力であるが，適切なインクルードファイルが与えられない場合，修正対象のソースファイルの解析が正しく行えない可能性がある．

作成ツールは入力が与えられると，それらに対して静的ソースコード解析を行い，4.3 節に従い修正必要箇所の特定を行う．特定した修正必要箇所は，4.5.2 節で述べる分類を行い，それぞれに応じた対応を行う．

作成ツールの出力は，修正後の対象ソースファイルと，作成したラッパー関数を記述したインクルード用ファイル，および作成ツールでは修正提案が行えない箇所のリストである．修正提案が行えない箇所については，4.5.2 節で述べる．

4.5.1 開発言語と使用ツール

作成ツールは Python 3.11.1[27] を用いて開発を行った．また，ソースファイルの解析の実行には，商用のソースコード解析ツール Understand[8] を用いて行っている．Understand での解析の実行には Python の Subprocess モジュール [28] を使用し，解析データの取り出しには Understand の Python API を使用している．

4.5.2 修正必要箇所の分類

4.2 節で述べたように，作成ツールの採用手法では修正が必要な箇所は関数呼び出し箇所と時刻を扱う変数の比較箇所の 2 つに大別される．しかし，作成ツールの実装を行う上では，静的解析のみを用いた自動での修正能力に限界があるため，比較箇所をさらに複数に分類し，行う対応を変えている．本節では，これを含めた修正が必要な箇所の分類について述べる．

作成ツールの実装においては，時刻を扱う変数の比較箇所を，以下の 3 つに分類している．

- 特定年情報比較箇所
- time_t 型値を比較する箇所
- その他の比較箇所

作成ツールにおいて修正対象ソースファイル中の修正必要箇所は，これらに関数呼び出し箇所を加えた計 4 種類に分類される．これらの分類は，修正方法の差異と，機械的修正の難易度の差異によって行っている．なお，特定年情報比較箇所とは，年情報を 2038 年問題に関わる値と比較する箇所を指す．ここでの 2038 年問題に関わる値とは，Unix 時刻の区切りとなる 1970 年や 2038 年と関わる値を表す．作成ツールにおいては，68，69，70，37，38，39，137，138，139 のいずれかの値と年情報との比較を，この分類に該当するものとしている．

また，分類の呼称とそれぞれの修正難易度を表 8 に示している．これらの修正難易度は，今回の実装方法に従って独自に判断したものであり，他の修正手法や実装方法を採った場合にも適用できるものではないことを記しておく．

4.5.3 修正対応の実施

表 8 に示したように，修正が必要な箇所の修正を自動的に行う難易度はその修正箇所の内容によって異なる．これを踏まえ，作成ツールでは分類した修正箇所の種別に応じて行う対応を分けている．

表 9 に 4.5.2 節での分類ごとの，作成ツールの対応内容を示している．

関数呼び出し箇所は，4.4.1 節に示したように，対応するラッパー関数の作成を別ファイルに行い，修正対象ファイルに対しては，呼び出しのラッパー関数呼び出しへの置き換えと，ラッパー関数ファイルのインクルード記述の追加を自動的に行う．

特定年情報比較箇所と time_t 値の比較箇所は，まとめて修正容易比較箇所としている．これらに対しては，4.4.2 節に基づいて自動で修正案を作成し，修正前の該当箇所のソースコードと作成した修正案をユーザーに提示する．ユーザーから承認または拒否の入力を受け付け，承認された修正案のみ対象ソースファイルへの反映を行う．

その他の比較箇所については，誤った修正が起きやすく自動的な修正を行うことが難しいため，修正困難比較箇所とし，該当箇所のソースコードをユーザーに提示するのみとしている．

4.6 評価

本節では，作成ツールの評価について述べる．作成ツールの評価には，FreeBSD[9] 12.1-RELEASE のソースコードを用いている．これは，作成ツールにて採用した手法を提案した [4] が手法の評価に FreeBSD 12.1-RELEASE を用いており，いくつかのソースファイルに対して，正解データとして扱える修正後ソースファイルが存在することから選択した．

表 8: 修正が必要な箇所の分類

分類	偽陽性率	修正パターン	自動修正難易度
関数呼出箇所	低	少	低
特定年情報比較箇所	中	少	中
time_t 値比較箇所	高	少	中
その他の比較箇所	高	多	高

表 9: 作成ツールの行う対応

対応による分類	内容による分類	対応内容
関数呼出箇所	関数呼出箇所	ラッパー関数を作成 ラッパー関数の呼び出しに置き換え
特定年情報比較箇所	修正容易比較箇所	修正案を作成 ユーザーの承認があれば修正実行
time_t 値比較箇所		
その他の比較箇所	修正困難比較箇所	該当箇所のリストを出力

表 10: 正常な実行が確認できたコマンド

ディレクトリ	コマンド名
src/bin	date，pax，ps，sh，sleep
src/sbin	adjkerntz，bectl，camcontrol，dhclient，dump，dumpfs，fsck_ffs，fsdb，growfs，hastd，ifconfig，init，iscontrol，nandfs，newfs，newfs_msdos，pfctl，ping6，restore，route，savecore，setkey，shutdown，sysctl，tunefs

はじめに，作成ツールが正常に実行可能であることを確認するため，FreeBSD 12.1-RELEASE の src/bin，src/sbin ディレクトリに含まれるコマンドのソースファイルに対して作成ツールを実行した．これらのディレクトリ内のコマンドのうち，時刻を扱う標準ライブラリである time.h をインクルードする記述を持つソースファイルが含まれるものを取り出して，ツールに入力として与える．これにより，表 10 に示す 30 のコマンドの計 194 ファイルに対して正常な実行完了が確認できた．これらのうち，37 のファイルで修正箇所が検出されている．これらのコマンドの他，src/sbin ディレクトリに含まれる ipfw コマンドにも，time.h をインクルードするソースファイルが存在するが，これを対象としたところ，実行時にエラーが発生し正常に完了できなかった．

4.6.1 先行研究との比較

作成ツールが採用手法を適切に実装できているか確かめるため，採用手法の提案を行った [4] における人力でのソースコード修正結果と，作成ツールによる修正結果の比較を行った．

[4] において人力で修正が行われているのは，FreeBSD 12.1-RELEASE[9] の date，stat，touch の 3 つのコマンドのソースファイルである．これらの修正結果を採用した修正手法における正解データとして，作成ツールを用いて修正を行ったソースファイルとの比較を行う．評価指標として用いるのは，修正を行った箇所のうち実際に修正が必要だったものの割合を表す適合率と，どれだけ漏れなく修正を行えたかを表す再現率の 2 つである．それぞれの定

表 11: 先行研究との比較結果

対象コマンド	先行研究	作成ツール	一致箇所数	適合率	再現率
date	6	13	5	0.38	0.83
stat	2	32	2	0.06	1.00
touch	6	10	6	0.60	1.00
合計	14	55	13	0.24	0.93

義は，数式 (1)，(2) に示している．

$$\text{適合率} = \frac{\text{作成ツールと正解データの双方で修正された箇所}}{\text{作成ツールによる修正箇所}} \tag{1}$$

$$\text{再現率} = \frac{\text{作成ツールと正解データの双方で修正された箇所}}{\text{正解データ中の修正箇所}} \tag{2}$$

なお，作成ツールによる修正では，4.5.3 における関数呼出箇所と修正容易比較箇所のみを反映し，修正困難比較箇所は除外している．これは，作成ツールの実装の段階において，修正困難比較箇所は誤修正が発生しやすく信頼性が低いものとして定めており，これを考慮すると適合率が大幅に低下することが想定されるためである．

以上の内容で先行研究と比較を行った結果を，表 11 に示す．表見出し中の “先行研究” および “作成ツール” は，それぞれ先行研究 [4] での修正箇所数と，作成ツールでの修正箇所数を表す．

結果として，3 つのコマンドのいずれに対しても，作成ツールによる修正は高い再現率を記録した．先行研究による修正箇所のうち，作成ツールによる修正箇所に含まれなかった箇所は，date コマンド中の 1 箇所のみであった．また，先行研究と作成ツールで一致した修正箇所では，修正内容が意味的に一致していることも確認できた．このことから，作成ツールは大江ら [3, 4] による 2038 年問題への対応手法に基づき，漏れのないソースコード修正を支援することが可能であると言える．

一方，作成ツールによる修正の適合率は，3 つのコマンドのいずれに対しても低い値となっている．このことは，作成ツールによる修正箇所には，実際には修正が不要である箇所が多数含まれることを表す．このため，作成ツールを用いた修正を行う際には，ツールが提案および実行する修正箇所が本当に修正の必要な箇所であるかを人力で確認する必要がある．

4.6.2 修正後コマンドの試行

作成ツールによる修正が 2038 年問題への対応手法として適切であるかを確かめるため，作成ツールを用いて修正したコマンドが，32bit システム上で 2038 年以降の時刻を正常に扱えるかの試行を行った．

図 7: date コマンドの試行結果

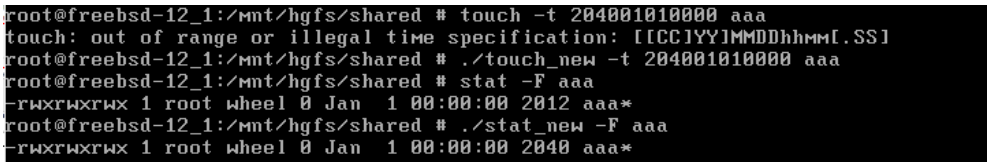


図 8: stat，touch コマンドの試行結果

対象としたのは，FreeBSD 12.1-RELEASE の date，stat，touch の 3 つのコマンドである．これらのコマンドに対して作成ツールを用いて修正を行った．この際，4.5.3 節における修正容易比較箇所の修正案の承認，拒否については，大江ら [4] による修正結果を参考に決定した．

これらのコマンドの修正前後それぞれのソースファイルを 32bit システム上でコンパイルし，実行した．結果，いずれのコマンドでも，修正を行う前は正常に実行できなかった，2038 年以降の時刻を扱う操作が，修正後のコマンドでは実行可能となった．

試行した際の画像を図 7，8 に示す．

図 7 では，date コマンドを修正した date_new コマンドを用いて 2040 年 1 月 1 日の時刻の出力を試みており，正常に出力されていることが確認できる．

また，図 8 では，はじめに修正前の touch コマンドと，修正を行った touch_new コマンドのそれぞれで，更新日時が 2040 年 1 月 1 日のファイルの作成を試みている．修正前のコマンドではエラーが発生しているが，修正後のコマンドではエラーが発生することなく実行できている．図 8 ではその後，修正前の stat コマンドと，修正を行った stat_new コマンドのそれぞれで，touch_new コマンドで作成した更新日時が 2040 年 1 月 1 日のファイルの情報の出力を試みている．修正前のコマンドではファイルの更新日時は 2012 年と想定と異なる値となっているのに対し，修正後のコマンドでは 2040 年と想定通りかつ touch_new コマンドと整合性のとれた値となっている．

4.7　考察

4.6 節の評価の結果から，作成ツールは大江ら [4] の 2038 年問題への対応手法に基づき，ソースコード中の修正が必要な箇所を高い再現率で特定し，適切な修正案を提示することができることが示された．これを利用することにより，従来は全て人力で行う必要があった，ソースコード中の修正が必要な箇所の特定と，それらに対する修正の実行の作業の省力化が可能である．

一方で，作成ツールが検出するソースコード中の修正が必要な箇所には，実際には修正が不要な箇所が多数存在する結果となった．これは，4.3 節で述べた修正必要箇所の特定手法の作成に際し，漏れのない修正を優先したことが一因と考えられる．適合率の低さは今後の課題ではあるものの，本研究の範囲では想定内の結果であると言える．

5　今後の課題

本節では，本研究で達成できなかった内容や，今後の課題となる内容について述べる．

5.1　2038 年問題に対するリスク調査

今回の調査では，時間的制約により，3.3.1 節の条件を満たすプロジェクトのうち，一部のみの調査しか行えていない．Suzuki ら [2] の調査では，より多くのプロジェクトに対して time_t 型の 64bit 化による不具合について調査を行っているが，それと同等の規模での符号付 32bit の time_t 型によるリスクの調査は存在しない．このため，今回の調査では対象にすることができなかったより多くのプロジェクトに対して同様の調査を行い，32bit リスクと 64bit リスクの双方について実態を調べることが，今後の課題となる．

また，今回の調査で検出したリスク箇所は，あくまでも不具合につながる可能性のある箇所であり，その全てが 2038 年以降不具合を引き起こすわけではない．より断定的に，確実に不具合につながる箇所のみを特定できるよう，ソースコード解析手法を改良することも，今後の課題として挙げられる．

5.2　2038 年問題に対するソースコード修正支援ツールの作成

4.6 節で述べたように，今回の作成ツールを FreeBSD[9] 12.1-RELEASE の ipfw コマンドのソースファイルに対して実行したところ，エラーが発生し正常に実行を完了できなかった．現在このエラーの原因は特定できていない．本研究で実行した範囲では，他のソースファイルに対してエラーが生じることはなかったため，このエラーが作成ツールを使用する上でどの程度影響を与えるかは未知数である．このエラーの原因を特定し解消するか，解消できない場合にはツールをより多くのソースファイルに対して実行したときにどの程度影響があるかを明らかにすることは，今後の課題となる．

また，4.6.1 節で述べたように，今回の作成ツールを用いたソースコード修正は，漏れのない修正の支援こそ可能であるが，適合率の低さ故に人力による誤修正の確認が欠かせないものとなっている．このため，作成ツールの適合率を向上させ，人力による確認の手間を減らすことも，今後の課題となっている．この課題の解決手法としては，プログラムの意味を考慮したソースコード解析を行うことや，再現率を維持しつつ適合率を向上できるような異なる修正必要箇所の特定方法を検討することが考えられる．

最後に，より多くの対象を用いての作成ツールの評価も，今後の課題として挙げられる．本研究においては，作成ツールによる修正の精度の評価は，大江ら [4] において修正が行われたソースファイルに対してのみしか行えていない．今後，より多くの対象を用いた評価を行い，作成ツールの性能を確認することや，そうした評価に使用する正解データとなる，人

力で 2038 年問題への修正対応が行われたソースコードを多数作成することにも取り組む必要がある.

6　まとめ

本研究では，GitHub[7] 上のソフトウェアを対象として，2038 年問題に対するリスクの存在調査を行った．調査を行った 172 のプロジェクトのうち，約 63%が 2038 年問題に対するリスクを 1 つ以上含むことが明らかとなった．さらに，検出されたリスクのうち，90%以上は time_t 型が符号付 32bit である環境におけるリスクであった．これらの結果から，2038 年問題による不具合は，多くのソフトウェアで発生する危険があり，特に time_t 型が符号付 32bit である環境においては，致命的な問題となる可能性があることがわかった．大規模な障害の発生を防ぐため，2038 年へ向けて，2038 年問題への適切な対応が進められる必要がある．

また，本研究では，2038 年問題への対応を容易にすることを目的として，大江ら [3] の手法に基づくソースコード修正支援ツールの作成を行った．作成ツールは，FreeBSD[9] 12.1-RELEASE の 30 のコマンドのソースファイルに対して正常に実行できることが確認された．また作成ツールは，FreeBSD 12.1-RELEASE の 3 つのコマンドのソースファイルにおいて，大江らの研究 [4] において人力で修正された修正箇所を，1 箇所を除き全て適切に修正することができた．さらに，作成ツールを用いて修正したこれらの 3 つのコマンドは，32bit システム上で，2038 年以降の時刻を正しく扱えることが確認された．これらから，作成ツールは大江ら [3] の手法を用いた 2038 年問題への対応の省力化に貢献することが可能であると言える．

リスク調査に関する今後の課題として，より多くのプロジェクトを対象として調査を行うことが挙げられる．また，より検出精度を高め，確実に不具合に繋がる箇所のみを検出する調査を目指すことも今後の課題となる．

ツール作成に関する今後の課題として，作成ツールによる誤検出の削減が挙げられる．作成ツールは，修正必要箇所を高い再現率で特定できているが，修正が不要な箇所の誤検出が多数発生しており，使用する際には人力での確認と誤検出箇所の除去が必要となっている．こうした確認に要する労力を削減するため，作成ツールを改良し，誤検出の削減を目指すことが今後の課題である．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 肥後芳樹 教授には，ご多忙の中，研究活動に対して多くの貴重な御助言や御指導を賜りました．心より深く感謝申し上げます．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠 准教授には，研究の着想から研究活動の直接の御指導，論文の執筆に至るまで，あらゆる場面で多くの御指導を賜りました．松下誠 准教授の適切な御指導により，本論文を完成させることができました．心より深く感謝申し上げます．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田哲也 助教には，発表の問題点の提示において，多くの御助言を賜りました．心より深く感謝申し上げます．

南山大学理工学部ソフトウェア工学科 井上克郎 教授には，研究の進め方や問題解決の手法などについて，様々な御助言を賜りました．心より深く感謝申し上げます．

最後に，その他様々な御指導，御助言等を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻肥後研究室の皆様に，心より深く感謝申し上げます．

参考文献

[1] time(3), FreeBSD 12.1-RELEASE Library Functions Manual (2003).

[2] Keita Suzuki, Takafumi Kubota, Kenji Kono: Detecting and Analyzing Year 2038 Problem Bugs in User-level Applications, 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC), pp65-74 (2019).

[3] 大江秀幸, 安藤友康, 松下誠, 井上克郎: 組込み機器開発における 2038 年問題への対応事例, 情報処理学会デジタルプラクティス Vol.10 No.3, pp.603–620 (2019).

[4] 大江秀幸, 松下誠, 井上克郎: 32bit Unix システムの 2038 年問題に対するプログラム修正法の提案, 情報処理学会論文誌, Vol.62 No.4, pp.1051-1055 (2021).

[5] Ryo Okabe, Jun Yabuki, Masakatsu Toyama: Avoiding Year 2038 Problem on 32-bit Linux by Rewinding Time on Clock Synchronization, 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp1019-1022 (2020).

[6] 水上陽向, 松下誠, 井上克郎: Unix の 2038 年問題に対する問題箇所特定ツール, ソフトウェアエンジニアリングシンポジウム 2021 論文集, pp275-282 (2021).

[7] GitHub, https://github.com/ （2023 年 2 月 1 日閲覧）.

[8] Understand^{TM} by SciTools, https://www.scitools.com/ （2023 年 2 月 1 日閲覧）.

[9] The FreeBSD Project, https://www.freebsd.org/ （2023 年 2 月 1 日閲覧）.

[10] Investigating why Steam started picking a random font, http://blog.pkh.me/p/35-investigating-why-steam-started-picking-a-random-font.html（2023 年 2 月 1 日閲覧）.

[11] 鈴木淳也: Windows 10 は全て 64bit になる 32bit から 64bit への完全移行は間もなく — ITmedia PC USER（2020-5-20）, https://www.itmedia.co.jp/pcuser/articles/2005/20/news062.html （2023 年 2 月 1 日閲覧）.

[12] 国立国会図書館 インターネット資料保存事業 — コンピュータ西暦２０００年問題 に関する報告書, 内閣コンピュータ西暦二千年問題対策室（2000-3-30）, https://warp.ndl.go.jp/info:ndljp/pid/284573/www.kantei.go.jp/jp/pc2000/contents.html（2023 年 2 月 1 日閲覧）.

[13] 横田隆夫：西暦 2000 年問題の意味と対応策，コンピュータソフトウェア，Vol.13, No.5,pp.412-419 (1996).

[14] US satellites safe after Y2K glitch — BBC News, http://news.bbc.co.uk/2/hi/americas/589836.stm （2023 年 2 月 1 日閲覧）.

[15] コンピュータ西暦２０００年問題の対応結果について— 電気事業連合会, https://www.fepc.or.jp/about_us/pr/kaiken/detail/200001-s1.html （2023 年 2 月 1 日閲覧）.

[16] 鈴木孝知, 中村建助: 「西暦 2038 年問題」でトラブル相次ぐ, 日経コンピュータ（2004-4-1）, http://tech.nikkeibp.co.jp/it/members/NC/ITARTICLE/20040325/1/ （2023 年 2 月 1 日閲覧）.

[17] 贄川俊: ２０年に１度ＧＰＳ時間リセット 前回は混乱，今回は？ — 朝日新聞デジタル（2019-4-6）, https://www.asahi.com/articles/ASM423RZ2M42UTIL00W.html （2023 年 2 月 1 日閲覧）.

[18] 「GPS 週数ロールオーバー」に関する重要なお知らせ＜バイク専用ポータブルナビゲーション＞ — Yupiteru, https://www.yupiteru.co.jp/corp/important/190405_bikenavi.html （2023 年 2 月 1 日閲覧）.

[19] 船舶用 GPS 受信機／航法装置のロールオーバー問題に関するお知らせ — 光電製作所, https://www.koden-electronics.co.jp/jpn/gps-rollover.html （2023 年 2 月 1 日閲覧）.

[20] 64 ビットになると何が変わる？――64 ビットプログラミングのデータモデル : 64 ビットコンピューティング最前線 — ITmedia エンタープライズ, https://www.itmedia.co.jp/enterprise/articles/0506/13/news006.html （2023 年 2 月 1 日閲覧）

[21] CTIME(3), FreeBSD 12.1-RELEASE Library Functions Manual (1999).

[22] PRINTF(1), FreeBSD 12.1-RELEASE Library Functions Manual (1999).

[23] Saving repositories with stars — GitHub Docs, https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars .

[24] WRITE(2), FreeBSD 12.1-RELEASE Library Functions Manual (1999).

[25] FREAD(3), FreeBSD 12.1-RELEASE Library Functions Manual (1999).

[26] SEND(2), FreeBSD 12.1-RELEASE Library Functions Manual (1999).

[27] Welcome to Python.org, https://www.python.org/ （2023 年 2 月 1 日閲覧）.

[28] subprocess — Subprocess management — Python 3.11.1 Documentation, https://docs.python.org/3/library/subprocess.html （2023 年 2 月 1 日閲覧）.