情報処理学会論文誌 Vol.65 No.9 1387–1398 (Sep. 2024)

32 bitを超えるtime_t型を持つ環境における2038年問題とその検出

星名 藍乃介[1,a)] 穐山 空道[2,b)] 上原 哲太郎[2,c)]

受付日 2023年12月21日, 採録日 2024年6月10日

概要：UNIX timeで表現されたタイムスタンプ値が32 bit符号付き整数型で定義された場合，2038年1月19日3時14分8秒以降の時刻で整数オーバーフローする．この整数オーバーフローはシステムにさまざまな不具合を引き起こす可能性があり，2038年問題と呼ばれる．2038年問題への対応として，64 bitなどの32 bitを超えるデータ型でタイムスタンプ値を扱う対策が一般に知られている．この対策において，タイムスタンプ値のデータ型は，外部プログラムも含めたデータフローの中で一貫して32 bitを超えるよう維持される必要がある．もし，どこか1カ所でも32 bit符号付き整数値として扱われると，整数オーバーフローを引き起こす可能性がある．本研究では，このような2038年問題のうち，ファイルシステムからの入出力時およびダウンキャスト時の整数オーバーフローに着目し，C言語プログラムから検出する手法を提案する．また，提案手法の有効性を示すため，検出手法に用いるツールを開発し，C言語OSSを対象に検証を行った．検証の結果，一部のFalse Positiveな検出は見られたものの，我々のツールは874のプロジェクトのうち294のプロジェクトから3,463の該当表現を検出した．

キーワード：2038年問題，ソースコード解析，リスク分析

The Year 2038 Problem in Environments with time_t exceeding 32 bits and its Detection Mechanism

Rannosuke Hoshina[1,a)] Soramichi Akiyama[2,b)] Tetsutaro Uehara[2,c)]

Received: December 21, 2023, Accepted: June 10, 2024

Abstract: When timestamp values expressed in UNIX time are defined as 32-bit signed integers, an integer overflow occurs at times after 3:14:08 on January 19, 2038. This integer overflow may cause various system failures and it is known as the 2038 problem. As a countermeasure against the 2038 problem, it is generally known the method to handle timestamp values with data types that exceed 32 bits, such as 64 bits. In this countermeasure, the data type of the timestamp value must be maintained to consistently exceed 32 bits in the data flow, including external programs. If the timestamp value is represented as a 32-bit signed integer type at even one point in the data flow, it may cause an integer overflow. In this study, we focus on integer overflows in file system I/O and downcast, and we propose a method to detect them from programs written in C. In addition, to demonstrate the effectiveness of the proposed method, we developed a tool used in the detection method and verified it on various OSS developed in C. As a result of the validation, our tool detected 3,463 relevant expressions from 294 projects out of 874 projects, although some false positives were found.

Keywords: Y2K38, source code analysis, risk analysis

[1] 立命館大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Ritsumeikan University, Ibaraki, Osaka 543–0001, Japan
[2] 立命館大学情報理工学部
College of Information Science and Engineering, Ritsumeikan University, Ibaraki, Osaka 543–0001, Japan

[a)] hoshina@cysec.cs.ritsumei.ac.jp
[b)] s-akym@fc.ritsumei.ac.jp
[c)] t-uehara@fc.ritsumei.ac.jp

1. はじめに

UNIX 系 OS のための標準規格である POSIX [1] では，UNIX time と呼ばれる 1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数で時刻を表現する方法が定められている．POSIX は C 言語のシステムコールやライブラリ関数を規定しており，その中で時刻表現形式として UNIX time が用いられている．POSIX が策定された当時，UNIX time は 32 bit 符号付き整数型で表現されることが一般的であった．現在でも，UNIX time を 32 bit 符号付き整数型であると暗黙のうちに仮定したプログラムが多く存在する．しかし，32 bit 符号付き整数型のタイムスタンプ値は，上限が $2^{31}-1$ であるため，2038 年 1 月 19 日 3 時 14 分 7 秒を過ぎると整数オーバーフローを引き起こす．この整数オーバーフローによってシステムにさまざまな不具合が発生する可能性があり，これを 2038 年問題と呼ぶ．POSIX が策定されて以降，UNIX time は UNIX 系 OS だけでなく，さまざまなシステムで広く使用されている．それらのうちのいくつかのシステムで，2038 年問題による不具合が生じると懸念される．2038 年問題は，影響範囲の広さや修正の困難さから深刻であると主張されている [2]．

2038 年問題への対応として，64 bit などの 32 bit を超えるデータサイズでタイムスタンプ値を扱う対策が一般に知られている．この対策において，タイムスタンプ値のデータサイズは外部プログラムも含めたデータフローの中で一貫して 32 bit を超えるよう維持される必要がある．もし，どこか 1 カ所でも 32 bit 符号付き整数値で扱われると，整数オーバーフローを引き起こす可能性がある．C 言語プログラムの場合，2038 年問題対策のために `time_t` 型を 64 bit で定義していても，32 bit 型へダウンキャストする実装がある場合は整数オーバーフローが発生しうる．また，既存手法では，このような 2038 年問題に起因する不具合を網羅的に検出ことが難しい．

そこで本研究では，`time_t` 型が 32 bit を超えるデータサイズで定義されているにもかかわらず起こる 2038 年問題について，網羅性高く検出することを目的とする．このような 2038 年問題のうち，ファイルシステムからの入出力時およびダウンキャスト時の整数オーバーフローに着目して提案を行う．さらに，提案手法に用いる検出ツールを開発し，普及率の高い OSS に対し検証を行うことで，本研究の提案手法の有効性および 2038 年問題の脅威性を示す．検証では，GitHub 上にアップロードされた Star 数の多い 874 の C 言語プロジェクトを対象に解析を行った．一部の False Positive な検出は見られたものの，我々のツールは 294 のプロジェクトから 3,463 の該当表現を検出した．なお，本研究は，著者らが過去に [3] で提案した方法を元に研究を発展させたものである．

本論文は次のように構成されている．2 章では，2038 年問題の背景や，既存の対策方法について説明する．3 章では，32 bit を超える `time_t` 型を持つ環境での 2038 年問題について説明する．4 章では提案手法，5 章では実装について説明する．6 章では，検出ツールを用いて実際に解析し，評価する．7 章では，本論文についてまとめる．

2. 背景

2.1 C 言語におけるタイムスタンプ値の型

C 言語においてタイムスタンプ値はさまざまな型で定義され，それらの型のデータサイズは環境依存である．次項でタイムスタンプ値の型として多く使用される `long` 型と `time_t` 型のデータサイズについて詳しく述べる．

2.1.1 `long` 型

`time_t` 型が登場する以前の C 言語プログラムにおいて，UNIX time の表現には，一般に `long` 型が用いられていた．実際に，`timeval` 構造体の `tv_sec` メンバは，`long` 型で定義されている環境が存在する．なお，`timeval` 構造体はタイムスタンプ値を指定するために使用され，`tv_sec` メンバは指定する時間の秒単位部分を表す．Win32 API では，`timeval` 構造体の `tv_sec` メンバが `long` 型として規格化されている [4]．また，`long` 型のデータサイズは C 言語の規格として定義されておらず環境依存である．参考文献 [5] によると，Windows 64 bit アプリケーションでの `long` 型は 32 bit である．

2.1.2 `time_t` 型

現在，UNIX time の表現には `time_t` 型が広く使われている．`time_t` 型の定義は規格化されておらず環境依存であり，`long` 型や `long long` 型，`double` 型などで定義されている．また，`long` 型や `long long` 型は環境によってデータサイズや符号が異なる．したがって，`time_t` 型は環境によってデータサイズや符号が異なる．

このように，タイムスタンプ値はさまざまな型で定義され，それらの型のデータサイズは C 言語の仕様として定められておらず，環境依存である．もし，32 bit 符号付き整数型で定義された `long` 型や `time_t` 型をタイムスタンプ値に使用すると，整数オーバーフローの可能性が生じる．

2.2 2038 年問題の対策

2038 年問題は，UNIX タイムスタンプを 32 bit 符号付き整数型で表現することに起因する．2038 年問題の対策として，32 bit を超えるデータサイズでタイムスタンプ値を扱う方法が一般に知られている．64 bit 符号付き整数型でタイムスタンプ値を扱うと，タイムスタンプ値の上限が $2^{63}-1$，すなわち，およそ西暦 3000 億年となる．このタイムスタンプ値を 64 bit として定義する方法で，さまざまなソフトウェアコミュニティでは 2038 年問題の修正作業が行われている．NetBSD のバージョン 6.0 以降 [6]，OpenBSD のバージョン 5.5 以降 [7]，Linux kernel のバージョン 5.6 以

降 [8] では，64 bit 環境だけでなく 32 bit 環境においても 64 bit で `time_t` 型を表現するよう変更された．Microsoft Visual Studio 2005 以降，C/C++における `time_t` 型のデフォルトの定義は，64 bit である `__time64_t` 型に変更された [9]．

3. 32 bit を超える time_t 型を持つ環境における 2038 年問題

2.2 節で述べたように，2038 年問題の対策として，UNIX time タイムスタンプを 32 bit を超えるデータサイズで定義する対策が一般的に知られている．この対策において，タイムスタンプ値のデータサイズは，外部プログラムを含めデータフロー全体で一貫して 32 bit を超える必要がある．もし，タイムスタンプ値がデータフローのどこかで 32 bit 符号付き整数型にダウンキャストされると，整数オーバーフローが発生する．C 言語で符号付き整数演算の整数オーバーフローは，未定義の動作である [10]．したがって，32 bit を超えるデータサイズでタイムスタンプ値を定義したにもかかわらず，2038 年問題に起因する不具合が発生する可能性が生じる．本研究では，タイムスタンプ値がデータフローのどこかで 32 bit 符号付き整数型にダウンキャストされるケースについて，以下の 2 つに分類した．

- プログラム外部で，タイムスタンプ値を 32 bit 符号付き整数型で定義している場合
- プログラム内部で，タイムスタンプ値を 32 bit 符号付き整数型にダウンキャストする場合

3.1 節および 3.2 節でそれぞれについて述べる．

3.1 プログラム外部での 32 bit 符号付き整数型のタイムスタンプの定義

ファイルシステムやデータベースなど，プログラム外部でタイムスタンプ値を 32 bit 符号付き整数型で扱うと，これらの外部モジュールで 2038 年問題に起因する整数オーバーフローの可能性が生じる．外部モジュールへの出力時に整数オーバーフローが生じたり，外部モジュールからの入力時にすでに整数オーバーフローした値を参照したりすると，C 言語プログラムで 32 bit を超える `time_t` 型としてタイムスタンプを使用していても，2038 年問題に起因する不具合が発生する可能性が生じる．次項で具体例を用いて紹介する．

3.1.1 ファイルシステムへの書き込み

ファイルシステムにファイルのタイムスタンプ情報を書き込む際に，整数オーバーフローが起こる場合がある．多くのファイルシステムでは，ファイルのタイムスタンプ情報として，最終アクセス時刻，最終修正時刻，最終属性変更時刻を保持している．ext2 や ext3，ReiserFS，Linux 5.10 より前の XFS といったファイルシステムでは，ファイルタイムスタンプ値のデータサイズを 32 bit 符号付き整数型で定義している．したがって，C 言語プログラムからファイルシステムが扱える日付範囲を超えたタイムスタンプ値を書き込むと，**図 1** のように，整数オーバーフローした値が格納される．ただし，Linux 5.10 以降の XFS ではタイプスタンプのデータサイズを拡張する対応がとられ，日付範囲が 1901 年 12 月から 2486 年 7 月まで拡大された．

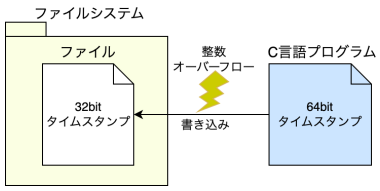

図 1 ファイルタイムスタンプ書き込み時の整数オーバーフロー
Fig. 1 Integer overflow when writing file timestamps.

3.1.2 データベース

いくつかのデータベースシステムでは，2038 年問題に起因する不具合が発生する可能性がある．MySQL では，時刻情報をサポートするデータ型である `TIMESTAMP` 型の日付範囲が，1970 年 1 月 1 日から 2038 年 1 月 19 日までである [11]．したがって，MySQL の `TIMESTAMP` 型が扱える日付範囲を超えたタイムスタンプ値が書き込まれると，整数オーバーフローが発生する．Microsoft SQL Server では，2 つの日付間の差を返す `DATEDIFF` 関数が提供されている [12]．`DATEDIFF` 関数の秒数を返す場合の返り値は，-2^{31} から $2^{31}-1$ の範囲の `int` 型であるため，整数オーバーフローした値が返却される可能性がある．

3.2 タイムスタンプ値の 32 bit 符号付き整数型へのダウンキャスト

プログラム内で，32 bit 符号付き整数型の最大値を超えたタイムスタンプ値を，32 bit 符号付き整数型にダウンキャストすると整数オーバーフローが起こる．たとえば，C 言語プログラムにおいて，64 bit `time_t` 型を 32 bit `int` 型や 32 bit `long` 型にキャストすると，整数オーバーフローが起こる場合がある．また，多くの環境で `int` 型は 32 bit であり [5]，2.1.1 項で示したように，`long` 型が 32 bit の環境は少なくない．したがって，64 bit `time_t` 型を `int` 型や `long` 型にキャストする処理は，多くの環境で整数オーバーフローを引き起こす可能性がある．整数オーバーフローした値はデータフローの中で参照され，予期せぬ動作を引き起こすことが想定される．

以下では，C 言語プログラムにおける「タイムスタンプ値の 32 bit 符号付き整数型へのダウンキャスト」について，キャスト方法に基づき分類する．キャストは，キャスト演算子を用いて行う明示的キャストと，互いに異なる型を用いた代入や演算をする場合にコンパイラが自動的に行う暗黙的キャストに分別できる．C 言語において，タイムスタンプ値を 32 bit 符号付き整数型に明示的，および暗黙的に

Listing 1 変数の代入時ダウンキャスト
```
int i = time(NULL);
```

Listing 2 関数宣言の return 時ダウンキャスト
```
int func(){
    time_t t = time(NULL);
    return t;
}
```

Listing 3 関数呼び出しの引数代入時ダウンキャスト
```
void func(int n) { /* ... */ }
time_t t = time(NULL);
func(t);
```

ダウンキャストにすることによって起こる整数オーバーフローについて具体例をあげる．

3.2.1 明示的なダウンキャスト

32 bit を超える `time_t` 型から 32 bit 符号付き整数型への明示的なダウンキャストとしては，`(int)time(NULL)` のように，キャスト演算子を用いて `time_t` 型の値をキャストする方法があげられる．

3.2.2 暗黙的なダウンキャスト

32 bit を超える `time_t` 型から 32 bit 符号付き整数型への暗黙的なダウンキャストは，「JIS X 3010 : 2003（ISO/IEC 9899 : 1999）」[13] より，以下の 3 パターンで起こる．

(1) 変数の代入時
(2) 関数の仮引数と実引数の型が異なるとき
(3) 関数宣言の返り値の型と return 文の式の型が異なるとき

(1) に関して，Listing 1 のように，`int` 型の変数に `time_t` 型の式を代入する場合に起こる．(2) に関して，Listing 3 のように，仮引数が `int` 型である関数に，実引数として `time_t` 型の値を渡す場合に起こる．(3) に関して，Listing 2 のように，返り値が `int` 型の関数宣言において，return 文に `time_t` 型の式を渡す場合に起こる．また，暗黙的なキャストとしては，四則演算やシフト演算時に起こる通常の算術変換もある．しかし，C 言語における通常の算術型変換の規則 [14] より，64 bit `time_t` 型と 32 bit `int` 型の算術演算では，`int` 型のオペランド側が 64 bit 拡張されるため，ダウンキャストは起こらない．

3.3 既存手法

3.1 節および 3.2 節であげた「32 bit を超える `time_t` 型を持つ環境における 2038 年問題」を検出する既存手法について述べる．C 言語の構文上，ダウンキャストは不正でないため，開発者が 2038 年問題に注意してソースコードをレビューしたとしても見逃しやすい．

そこで，コンパイラの警告によってダウンキャスト可能性の有無を確認する方法が考えられる．いくつかのツールでは，64 bit のデータ型から 32 bit の型への暗黙的なキャストを警告する機能がある．たとえば，Clang の場合，`-Wshorten-64-to-32` オプションを指定することで，64 bit 整数型から 32 bit 整数型への暗黙的なダウンキャストが警告される．すべての暗黙的なダウンキャストを修正するという方法も考えられるが，特に暗黙的なダウンキャストが多く存在するプロジェクトにおいて，そのすべてを修正することは困難である．したがって，2038 年問題に関係するダウンキャストの発見においては，`time_t` 型からのダウンキャストのみ出力されることが望まれる．しかし，`-Wshorten-64-to-32` オプションでは，64 bit から 32 bit の型へのすべてのキャストが警告されるため，2038 年問題固有のダウンキャストを検出することは難しい．また，明示的なダウンキャストについては，既存のコンパイラで検出することは難しい．たとえば，Clang ですべての警告を表示するオプションである `-Wall/-Weverything` を使用しても，`(int)time(NULL)` のようなコードに対しては警告されない．Clang が明示的なダウンキャストによる整数オーバーフローを警告しない理由は，コンパイラが整数オーバーフローを開発者の意図した動作として解釈するためである．

鈴木らの研究 [15] では，`time_t` 型が 64 bit 符号付き整数型で定義されている C 言語プログラムにおける，2038 年問題に起因するバグを検出する手法が提案されている．この手法では，タイムスタンプを返す関数を起点としたデータフローを解析し，さらにコンパイラの警告によってバグを発見する．データフローの起点となった関数は `time()`，`mktime()`，`gettimeofday()` であるが，タイムスタンプ値を返す関数はこれら以外にも存在する．たとえば，`stat()` から得られる `st_atime` などは，鈴木らの研究 [15] では解析の対象外である．`st_atime` を含め任意のタイムスタンプ値についても評価するために，任意のタイムスタンプ値を網羅的に検査する手法の確立が求められる．また，3.1 節で述べたように，プログラムレイヤで `time_t` 型を 64 bit で定義していたとしても，ファイルシステムなど外部モジュールでタイムスタンプ値を 32 bit 符号付き整数型として扱うと，タイムスタンプ値の書き込み時に整数オーバーフローする場合がありうる．このような，タイムスタンプ値を 32 bit 符号付き整数型で扱う外部モジュールへ出力するときの整数オーバーフローについても，検出手法の確立が求められる．

静的解析ツール Coverity [16] では，`Y2K38_SAFETY` という，C 言語における 2038 年問題を検出するためのチェッカーが存在する．Coverity の `Y2K38_SAFETY` は有力な解析ツールであり，実際に GitHub Code Search で `Y2K38_SAFETY` と検索するといくつかのプロジェクトで Coverity による警告ログが確認できた．タイムスタンプ値を 32 bit 符号付

き整数型で扱う外部モジュールへの出力時の整数オーバーフローに関しても，Coverity の Y2K38_SAFETY は対象外である．

このように，既存手法の対象外となる 2038 年問題が多く存在することから，より網羅性の高い検出手法の確立が求められる．

我々は先行研究 [3] において，3.2 節で述べたようなタイムスタンプ値の 32 bit 符号付き整数型へのダウンキャスト，および 3.1.1 項で述べたようなファイルシステムへの書き込み時の整数オーバーフロー可能性について，検出する手法を提案した．しかし，先行研究の検出手法では以下のような問題点があった．

(1) 64 bit time_t 型から 32 bit long 型への暗黙的なダウンキャストを検出できない点
(2) ファイルタイムスタンプを変更する関数にはさまざまな種類が存在するが，それらを網羅していない点

本研究では，先行研究の問題点を修正した検出手法を提案する．

4. 提案手法

3 章では，time_t 型が 32 bit を超えるデータサイズで定義されていたとしても，プログラム外部でタイムスタンプ値を 32 bit 符号付き整数型で定義する場合や，プログラム内部でタイムスタンプ値を 32 bit 符号付き整数型にダウンキャストする場合，2038 年問題に起因する不具合が発生する可能性があると述べた．2038 年問題への対応として，このような不具合の可能性があるコードを，C 言語ソースコードから発見する必要がある．ここではこのようなソースコード上の特定の命令列を，expression にちなみ「表現」と呼ぶこととする．また，3.3 節で述べたように，既存手法を用いても，32 bit を超える time_t 型から 32 bit 符号付き整数型へのダウンキャスト表現を網羅的に発見することは難しい．

そこで，本研究では，time_t 型が 32 bit を超えるデータサイズで定義されているにもかかわらず起こる 2038 年問題に対して，より網羅性の高い検出手法を提案する．本提案手法では，以下の点から既存手法で対象外であった表現も含めて検出できる．

- 「キャスト式の発見」と「解析環境の time_t 型の定義の変更」によって time_t 型から 32 bit 符号付き整数型へのダウンキャストを網羅的に検出する．
- ファイルタイムスタンプを書き込む標準ライブラリ関数を網羅している．

検出手法では，検出対象となる抽象構文木のパターンを定義し，入力ソースコードからこれに合致するコードを探索することで検出を試みる．

4.1 検出対象

以下に，検出対象の表現を示す．

(1-a) ファイルタイムスタンプの書き込み表現
(1-b) ファイルタイムスタンプの参照表現
(2-a) 64 bit time_t 型から 32 bit int 型へのダウンキャスト表現
(2-b) 64 bit time_t 型から 32 bit long 型へのダウンキャスト表現

4.1.1 ファイルタイムスタンプの読み書き表現

検出対象 (1-a)，(1-b) について述べる．C 言語プログラムでファイルタイムスタンプを変更する方法はいくつか存在するが，今回は標準ライブラリ関数に着目する．The Linux man-pages project [17] または Debian Manpages [18] に掲載されている標準ライブラリ関数に限定して調査すると，utime，utimes，utimensat，futimes，futimens，futimesat，lutimes が該当した．

整数オーバーフローしたファイルタイムスタンプをプログラム内部から参照すると，予期せぬ動作を引き起こすことが想定される．C 言語では，標準関数の stat 関数や lstat 関数，fstat 関数を使用して，stat 構造体からファイルのタイムスタンプを取得できる．stat 構造体のメンバには，ファイルデータの最終アクセス時刻を表す st_atime，ファイルデータの最終修正時刻を表す st_mtime，ファイルデータの最終属性変更時刻を表す st_ctime がある．

4.1.2 time_t 型から int 型・long 型へのダウンキャスト表現

検出対象 (2-a)，(2-b) について述べる．3.2 節で述べたように，多くの環境で int 型は 32 bit 符号付き整数型で定義されるため，time_t 型から int 型へのダウンキャスト表現を検出対象とする．また，2.1.1 項で述べたように long 型が 32 bit 符号付き整数型で定義される環境も存在するため，time_t 型から long 型へのダウンキャスト表現も検出対象とする．

4.2 検出手法で用いるデータ構造

検出手法が対象とするデータ構造には，変数，式，関数，構造体およびそのメンバ，それぞれの型情報が必要となる．これらを含むデータ構造として，抽象構文木があげられる．抽象構文木は，プログラムの各要素を木構造として表したデータ構造である．抽象構文木の葉はノードと呼ばれ，ノードには型情報や文字列などの情報が付与されている．抽象構文木はコンパイラの内部でプログラムの中間表現として使用されることが多く，そのため，抽象構文木を対象としたソースコード解析ツールも多く存在する．本研究では，既存の解析ツールを活用して実装を容易にするために，対象となるプログラムを抽象構文木に変換して検出手法を適用する．

4.3 検出手順

以下の手順で検出対象の表現を発見する．

(1) 各検出対象の表現に対応する抽象構文木のパターンを決める．

(2) 検出ツールが，検出対象の抽象構文木のパターンに合致する抽象構文木ノードを探索する．

(3) 解析者が，発見された抽象構文木ノードが特定の環境下で整数オーバーフローを引き起こしうる表現であると判断する．

(1) について，4.4 節および 4.5 節で抽象構文木のパターンを述べる．(2) について，ソースコードの中から，各検出対象に対応する抽象構文木ノードを探索する．(3) について，各検出項目が 2038 年問題となる環境について述べる．ファイルタイムスタンプの書き込み，および参照表現が発見された場合，ファイルタイムスタンプを 32 bit 符号付き整数型で扱うファイルシステムを使用する環境で，整数オーバーフローを引き起こしうると判断する．`time_t` 型から `int` 型へのキャスト表現が発見された場合，`time_t` 型が 64 bit，`int` 型が 32 bit で定義されている環境で，整数オーバーフローを引き起こしうると判断する．`time_t` 型から `long` 型へのキャスト表現が発見された場合，`time_t` 型が 64 bit，`long` 型が 32 bit で定義されている環境で，整数オーバーフローを引き起こしうると判断する．

また，手順の最後には，ツールが False Positive な検出を行う可能性を考慮して，解析者が検出結果を直接確認する．False Positive な検出としては，`time_t` 型から 32 bit 符号付き整数型へのダウンキャスト表現検出において，たとえば以下のようなパターンが考えられる．

- UNIX タイムスタンプではない `time_t` 型の利用
- ダウンキャストはしているが，変数のとりうる値から明らかに整数オーバーフローが発生しないもの

解析者は，このような False Positive が発見された場合，それらを検出結果から排除する．

4.4 ファイルタイムスタンプの読み書き表現の検出

本研究では，ファイルタイムスタンプを書き込む処理のうち，4.1.1 項で紹介した関数に限定して検出する．ファイルのタイムスタンプの書き込み表現に対応する抽象構文木のパターンを以下に示す．

- 標準ライブラリの utime.h で定義されている `utime` 関数の呼び出し式
- 標準ライブラリの sys/time.h で定義されている `utimes` 関数の呼び出し式
- 標準ライブラリの sys/stat.h で定義されている `utimensat` 関数の呼び出し式
- 標準ライブラリの sys/time.h で定義されている `futimes` 関数の呼び出し式
- 標準ライブラリの sys/stat.h で定義されている `futimens` 関数の呼び出し式
- 標準ライブラリの sys/time.h で定義されている `futimesat` 関数の呼び出し式
- 標準ライブラリの sys/time.h で定義されている `lutimes` 関数の呼び出し式

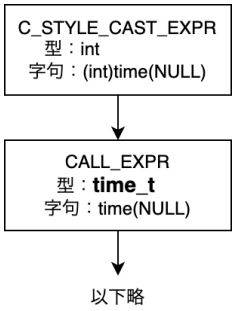

図 2 `(int)time(NULL)` の抽象構文木
Fig. 2 Abstract syntax tree for `(int)time(NULL)`.

ファイルタイムスタンプの参照表現としては `stat` 関数の式などがあるが，`stat` 関数はファイルタイムスタンプ以外のファイルの状態を取得するときにも使用される．そこで，本研究ではファイルタイムスタンプを保持する `stat` 構造体メンバの式に限定して考える．ファイルタイムスタンプの参照表現に対応する抽象構文木のパターンは，`stat` 構造体メンバの `st_atime`，`st_mtime`，`st_ctime` の式とする．これらのパターンに合致する抽象構文木ノードを探索することで，ファイルタイムスタンプへの書き込み，および参照表現を検出する．

4.5 `time_t` 型から `int` 型および `long` 型へのキャスト表現の検出

`time_t` 型から `int` 型または `long` 型へのキャスト表現に対応する抽象構文木のパターンを以下に示す．

- `int` 型のキャスト式のうち，キャスト元の型が `time_t` 型である式
- `long` 型のキャスト式のうち，キャスト元の型が `time_t` 型である式

たとえば，`(int)time(NULL)` という明示的なキャスト式を抽象構文木に変換すると，**図 2** のようになる．`C_STYLE_CAST_EXPR` は明示的なキャスト式を意味するノード，`CALL_EXPR` は，関数呼び出しを意味する式である．ここで，`time_t` 型の `C_STYLE_CAST_EXPR` および `int` 型の `CALL_EXPR` をみることで，`time_t` 型から `int` 型への明示的なキャスト表現であると判断できる．

4.5.1 `time_t` 型由来の型の判定

キャスト元の型が `time_t` 型であるかを判定する手法について考える．図 2 に示す `(int)time(NULL)` の抽象構文木の場合は，キャスト式ノードの子ノードの型情報を用いて，キャスト元の型が `time_t` 型であるかを判定できる．一方で，**図 3** に示す `(int)(1+time(NULL))` の抽象構文

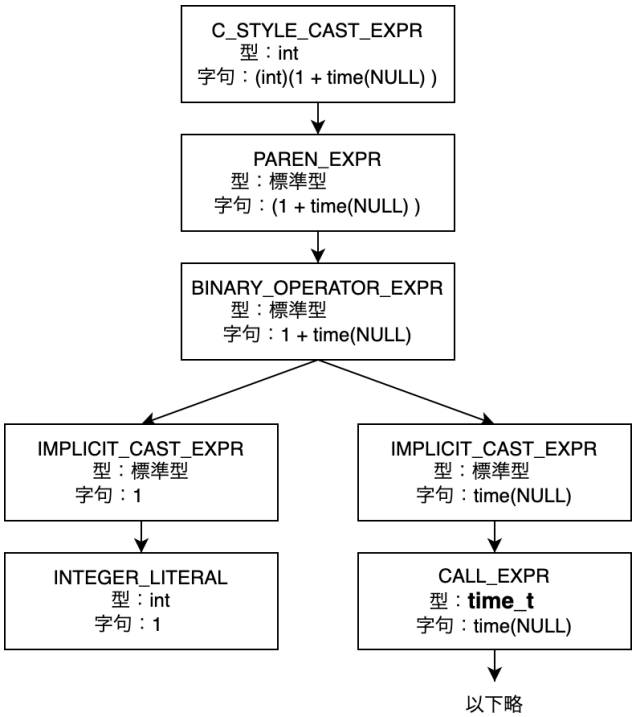

図 3 `(int)(1+time(NULL))` の抽象構文木
Fig. 3 Abstract syntax tree for `(int)(1+time(NULL))`.

木において，キャスト式ノードの子ノードは `long` 型や `long long` 型などの標準符号付き整数型（以下，標準型とする）になる．この場合，子ノードの型情報だけでは，`(int)(1+time(NULL))` が `time_t` 型の式に由来する標準型であるか否かを判定できない．このように `time_t` 型に由来する型を「`time_t` 型由来の型」と呼ぶこととする．

ある式ノードが `time_t` 型由来の型を持つか否かを判定するためには，子孫ノードまで探索して型情報を確認する必要がある．本提案では，ある式ノードの子ノードを再帰的に探索することで，ある式ノードが `time_t` 型由来の型であるかを判定する．その判定アルゴリズムを「`time_t` 型由来判定関数」とし，Algorithm 1 に示す．

この「`time_t` 型由来の型」の判定は，入力に与えられた式ノードの数を N_V として $O(N_V)$ 時間で解ける．したがって，「`time_t` 型由来の型」の判定にかかる処理時間は，解析対象のソースコードの規模に比例する．

4.5.2 解析環境の `time_t` 型の定義

`time_t` 型から `long` 型へのキャスト表現の検出手法を実装する際の注意点として，解析環境の `time_t` 型の定義を考慮する必要がある．たとえば，Ubuntu などの環境では `time_t` 型が `long` 型で定義されているが，このような環境では `time_t` 型から `long` 型への暗黙的なキャストは起こらない．その場合，`time_t` 型から `long` 型への暗黙的キャスト表現を検出できない．そこで，本研究では，検出ツールが参照する `time_t` 型の定義ファイルで，`typedef long long time_t` などのように，`time_t` 型を `long` 型以外の型で定義するよう変更する．

Algorithm 1 `time_t` 型由来判定関数

Input: V - 式ノード
Output: V が `time_t` 型由来である場合は `true`，そうでない場合は `false`

```
function is_timet_or_similar(V)
    if V の型情報が time_t 型 then
        return true
    end if
    if V ∈ 二項演算式ノード then
        V_l = V の左項のノード
        V_r = V の右項のノード
        return is_timet_or_similar(V_l) ∨ is_timet_or_similar(V_r)
    end if
    if V ∈ paren ノード then
        V_c = V の子ノード
        return is_timet_or_similar(V_c)
    end if
    return false
end function
```

5. 実装

4 章では，`time_t` 型を 32 bit を超えるデータサイズで定義しているにもかかわらず起こる 2038 年問題に対して，抽象構文木を用いて検出する手法を提案した．本研究では，提案手法を Clang Static Analyzer として実装した．本検出ツールは，以下の環境，およびソフトウェアで実装した．

- プロセッサ：Intel Xeon(R) CPU E3-1225 v6
- OS：Ubuntu 20.04.6 LTS（64 bit）
- メモリ：62.7 GiB
- LLVM/Clang 11.0.0

5.1 検出ツールの外部仕様

本検出ツールは，Clang Plugin として Clang コンパイラによる静的解析時に実行される．検出対象として，本検出ツールは，4.4 節および 4.5 節で述べた抽象構文木ノードを探索する．入力として，本検出ツールは解析対象の C 言語ファイルのパスを受け取る．検出結果について，本検出ツールは以下を標準出力する．

- 4.1 節で述べた検出対象の種類
- ファイルパス
- 該当箇所の行および列

5.2 処理の流れ

処理の流れを**図 4** に示す．本検出ツールの入力に C 言語のソースコードを与えると，構文解析し，抽象構文木を生成する．その後，抽象構文木から検出対象のパターンにマッチするノードを探索し，見つかった場合は通知する．

抽象構文木から検出対象のパターンを探索する処理について述べる．本検出ツールは，Clang の AST Matcher という Clang の抽象構文木に対してパターンマッチを行うためのツールを用いて，検出対象の抽象構文木ノードを検索

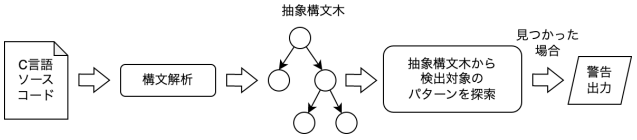

図 4 処理の流れ

Fig. 4 Overall procedure of proposed method.

する．ファイルタイムスタンプの参照表現の探索については，4.4 節で述べた抽象構文木ノードの条件にマッチする AST Matcher を作成し，検出ツールに組み込んだ．また，`time_t` 型から `int` 型および `long` 型へのキャスト表現については，以下の手順で探索する．

(1) AST Matcher で，キャスト先が `int` 型および `long` 型であるキャスト式を検索する．
(2) マッチしたキャスト式のキャスト元の型が `time_t` 型かを，4.5.1 項で述べた `time_t` 型由来判定関数を用いて判定する．

6. 検証

6.1 検証方法

5 章で実装した検出ツールを用いて，ファイルタイムスタンプおよびダウンキャストに起因する 2038 年問題が，実際に脅威になりうることを示す．2038 年問題の影響は，人気で使用数の多いソフトウェアほど大きいと考えられる．そこで，GitHub の Star 数を指標として検証する．GitHub REST API を用いて言語設定が C 言語のプロジェクトを検索し，2023 年 11 月 8 日現在で Star 数が最も多い 874 プロジェクトを収集した．プロジェクトの Star 数は，最も多いもので約 16 万 Star，最も少ないもので約 1,300 Star であった．収集した各プロジェクトの C 言語ファイルを入力に，我々の検出ツールを実行した．**表 1** に解析したプロジェクトを示す．各プロジェクトの解析にかかった検出ツールの実行時間について，**図 5** に示す．最もソースコードの行数が多かった約 3,300 万行のプロジェクトでも，約 1 時間という実用的な時間で解析が完了した．

6.2 `time_t` 型のダウンキャスト表現の検出結果

我々の検出ツールは，合計 1,947 件の `time_t` 型のダウンキャスト表現を検出した．プロジェクトごとの検出件数を**図 6**，**図 7** に示す．`time_t` 型から `int` 型へのダウンキャストは 173 プロジェクトで発見された．`time_t` 型から `long` 型へのダウンキャストは 112 プロジェクトで発見された．

発見された典型的なパターンを以下に示す．

(1) 現在時刻をダウンキャストするパターン
(2) ダウンキャストした値を外部に出力するパターン
(3) プログラム外部からの時刻データをダウンキャストするパターン

表 1 解析したプロジェクト

Table 1 Analyzed projects.

解析ファイル数	698,326 ファイル
解析プロジェクト数	874 プロジェクト
一件でも検出されたプロジェクト数	294 プロジェクト

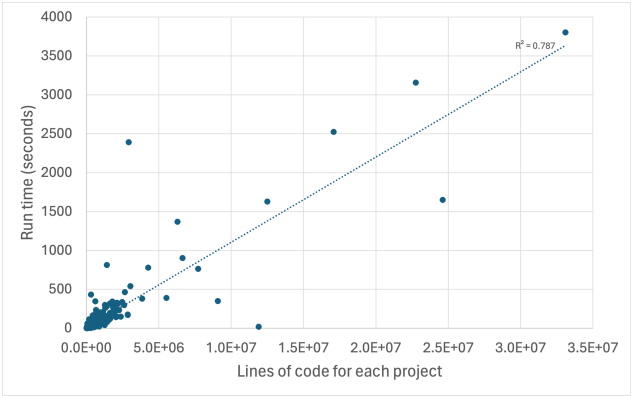

図 5 検出ツールの実行時間

Fig. 5 Run time of proposed detection tool.

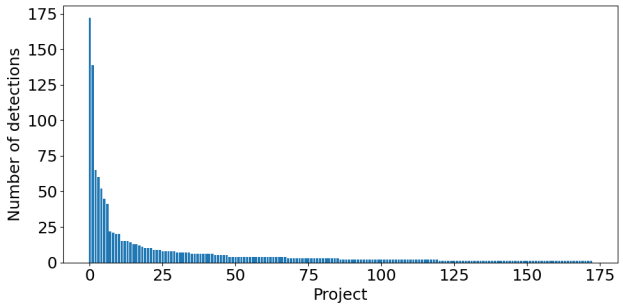

図 6 `time_t` 型から `int` 型へのダウンキャストのプロジェクトごとの検出数

Fig. 6 Number of detected downcast cases from `time_t` to `int` type per project.

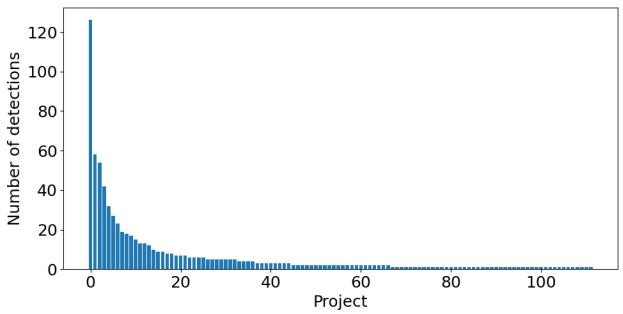

図 7 `time_t` 型から `long` 型へのダウンキャストのプロジェクトごとの検出数

Fig. 7 Number of detected downcast cases from `time_t` to `long` type per project.

まず，(1) は，`time()` 関数などを用いて現在時刻を取得し，その結果を int 型や long 型にダウンキャストしているパターンである．たとえば，Mirai-Source-Code [19] というプロジェクトで発見された Listing 4 は，現在時刻の 1 秒後を表すタイムスタンプを，int 型の構造体メンバに代

Listing 4 現在時刻をダウンキャストするパターン

```
struct connection {（中略）

    struct {
        char data[512];
        int deadline;
    } output_buffer;（中略）

};
struct connection *conn;
conn->output_buffer.deadline = time(NULL) + 1;
```

Listing 5 ダウンキャストした値を外部に出力するパターン（int 型）

```
snprintf(tmpfile,256,
    "temp-%d.%ld.rdb",
    (int)server.unixtime,(long int)getpid());
```

Listing 6 ダウンキャストした値を外部に出力する例（long 型）

```
snprintf(tbuf, sizeof(tbuf), "%ld.%03ld s",
    (long) tv.tv_sec,
    (long) (tv.tv_usec / 1000));
```

入している．さらに，構造体名や周辺のコードから，この構造体メンバは出力が完了するまでの最大時間を指定するためのものであると考えられる．もし，int 型が 32 bit 符号付き整数型の環境でこのコードが動作すれば，`deadline` メンバに整数オーバーフローした値が代入され，意図せぬ動作を起こす可能性がある．

次に，(2) はダウンキャストした値をログや HTTP ヘッダーなどの属性値として，外部環境に出力しているパターンである．たとえば，AnnotatedCode [20] というプロジェクトで発見された Listing 5 は，サーバの UNIX タイムスタンプを `int` 型でダウンキャストし，外部環境に出力している．agensgraph [21] というプロジェクトで発見された Listing 6 は，`timeval` 構造体の `tv_sec` メンバを `long` 型でダウンキャストし，外部環境に出力している．もし，int 型や `long` 型が 32 bit 符号付き整数型の環境でこれらのコードが動作すれば，整数オーバーフローした値を外部環境に出力する可能性がある．

最後に，(3) は，HTTP リクエストヘッダーのタイムスタンプ属性やファイルタイムスタンプといった，プログラム外部環境からのタイムスタンプをダウンキャストしているパターンである．本検証により発見された例として，openwrt-packages [22] というプロジェクトで発見されたコードを Listing 7 示す．このダウンキャストは，3.2.2 項で述べた「関数の仮引数と実引数の型が異なるとき」に該当する．このコードは，HTTP リクエストヘッダーの `If-Modified-Since` 属性の値と，ファイルの最終更新日時とを比較する用途で使用されていると考えられる．もし，int 型が 32 bit 符号付き整数型の環境でこのコードが動作すれば，整数オーバーフローを引き起こし，意図せぬ動作を起こす可能性がある．

Listing 7 プログラム外部からの時刻データをダウンキャストする例

```
int _httpd_checkLastModified(
    request *r, int modTime){
    /* 中略 */
}
/* 中略 */
if (stat(path, &sbuf) < 0){
    _httpd_send404(server, r);
    return;
}
if (_httpd_checkLastModified(
    r, sbuf.st_mtime) == 0){
    /* 中略 */
}
```

以上の検証結果から，`time_t` 型のダウンキャストによって，2038 年問題に起因する不具合の可能性があることが示された．

6.3 ファイルタイムスタンプの読み書き表現の検出

我々のツールは，合計で 1,516 件のファイルタイムスタンプの読み書き表現を検出した．ファイルタイムスタンプの書き込み表現は 88 プロジェクト，ファイルタイムスタンプの参照表現は 138 プロジェクトで発見された．プロジェクトごとのファイルタイムスタンプの書き込み表現の検出数を**図 8** に，ファイルタイムスタンプの参照表現の検出数を**図 9** に示す．

litetree [23] というプロジェクトで発見された Listing 8 は，`utimes` 関数を用いてファイルタイムスタンプに現在時刻を書き込んでいる．もし，このコードが ext3 などのファイルシステムで 2038 年以降に動作した場合，ファイルタイムスタンプを書き込む際に整数オーバーフローが発生する可能性がある．

STAR [24] というプロジェクトで発見された Listing 9 は，`stat` 関数を用いてファイルのタイムスタンプ値を参照し，`if` 文の条件式に使用している．もし，このコードが ext3 などのファイルシステムで動作し，参照したファイルタイムスタンプが整数オーバーフローしていれば，意図せぬ動作を起こす可能性がある．

6.4 False Positive

検証結果からファイルタイムスタンプの読み書き表現の誤検出は発見されなかった．一方で，ダウンキャスト表現の検出結果に関しては以下のような False Positive が発見された．

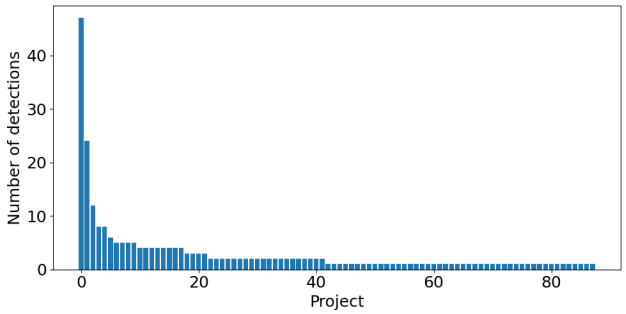

図 8 ファイルタイムスタンプの書き込み表現のプロジェクトごとの検出数

Fig. 8 Number of detected downcast cases per project where file timestamps is written.

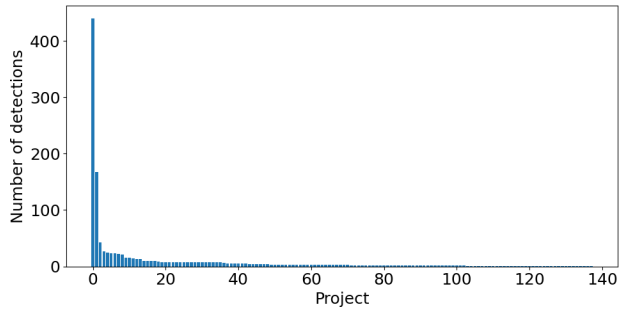

図 9 ファイルタイムスタンプの参照表現のプロジェクトごとの検出数

Fig. 9 Number of detected downcast cases per project where file timestamps is read.

Listing 8 ファイルタイムスタンプの書き込み

```
struct timeval times[2];
/* 中略 */
times[0].tv_sec = time(0);
/* 中略 */
if( utimes(zFile, times) ){
  return 1;
}
```

Listing 9 ファイルタイムスタンプの参照表現の検出例

```
struct stat stat_idx,stat_main;
if ( !stat(fn, &stat_main)
        && !stat(fnidx, &stat_idx)){
  if (stat_idx.st_mtime < stat_main.st_mtime)
    fprintf(stderr, "Warning: // 中略
}
```

(1) UNIX タイムスタンプとしての用途ではない `time_t` 型のダウンキャスト

(2) 値のとりうる範囲から明らかに整数オーバーフローしないと考えられるダウンキャスト

(1) に関して，UNIX エポックからの経過ミリ秒数や，ID として，タイムアウト秒数，timeval の差分をとるなどの用途で使用している実装が発見された．たとえば，polipo [25] というプロジェクトで発見された Listing 10 は，2 つの `time_t` 型変数の差分を `int` 型の変数にキャストしている．この差分の結果は UNIX タイムスタンプではないと考えられ，その場合，ダウンキャスト部分を検出して修正することは過度な修正であり，False Positive となる．このような False Positive を自動的に検出するには，変数や式の用途の分析が必要であり，一般には技術的に困難である．一方で，たしかに検出結果としては False Positive であるが，修正作業においては修正必要性の有無にかかわらず修正してもよいと考えられる．たとえば，Listing 10 の修正作業において，標準出力のデータサイズを増やしてダウンキャストしないよう修正したとしても，プログラムは期待どおり機能する．以上より，False Positive な検出であるものの，修正作業においては実用上の問題は大きくないと考えられる．

Listing 10 `timeval` の差分

```
n = snnprintf(connection->buf, n, bufsize,
    "\r\nAge: %d",
    (int)(current_time.tv_sec - object->age));
```

(2) に関して，`time_t` 型変数の割算または剰余算の結果を `int` 型変数に代入する実装が発見された．たとえば，Listing 11 の最後の行は，`time_t` 型の割算の結果を `int` 型の変数に代入している．割算結果のとりうる範囲から，このダウンキャストでは整数オーバーフローしない．その場合，このようなコードは 2038 年問題対応の対象とはならない．修正しなくてもよいコードが検出されることは修正コストの増大を招くため，提案手法の改善が求められる．改善案としては，本研究での提案のような型レベルの検査ではなく，区間演算やデータフロー解析といった値のとりうる範囲を考慮した解析が必要である．一方で，たしかに検出結果としては False Positive であるが，修正作業においては修正必要性の有無にかかわらず修正してもよい．たとえば，Listing 11 の修正作業において，代入先のデータサイズを増やしてダウンキャストしないよう修正したとしても，プログラムは期待通り機能する．以上より，False Positive を排除するような手法の改善は求められるものの，修正作業においては実用上の問題は大きくないと考えられる．

6.5 考察

我々の調査により，GitHub の Star 数上位 874 プロジェクト中 284 プロジェクトで潜在的に 2038 年問題の脅威を有する記述が見つかった．GitHub の Star 数上位のプロジェクトは，それを利用したソフトウェアが世の中に多く存在することを意味すると考えられる．そのため，`time_t`

Listing 11 割算の結果を int 型に代入

```
void nolocks_localtime(
        struct tm *tmp, time_t t,
        time_t tz, int dst) {
/* 中略 */
const time_t secs_hour = 3600;
/* 中略 */
time_t seconds = t % secs_day;
/* 中略 */
tmp->tm_hour = seconds / secs_hour;
```

型を 64 bit に移行したとしても，2038 年問題を引き起こすソフトウェアは潜在的に多く存在することが懸念される．

time_t 型から int 型へのダウンキャスト表現は 1225 件，time_t 型から long 型へのダウンキャスト表現は 722 件検出された．C 言語において整数オーバーフローは未定義の動作である [10] ため，検出されたダウンキャスト表現によって整数オーバーフローが生じれば，プログラム動作が変更される可能性が高いと考えられる．

ファイルタイムスタンプの読み書き表現の検出に関して，書き込み表現は 255 件，参照表現は 1261 件検出された．仮に，ext3 などのファイルシステムでこれらのコードが動作した場合，2038 年問題に起因する不具合が発生する可能性が高いと考えられる．

3.3 節で述べた既存検出手法との比較について述べる．コンパイラの警告による検出手法では，Listing 5 のような明示的なダウンキャストを検出することは難しい．鈴木らの研究 [15] の検出手法では，Listing 7 のような stat 関数から取得されたタイムスタンプをダウンキャストするコードは検出対象外である．Coverity による検出では，Listing 8 のようなファイルタイムスタンプ書き込み時に整数オーバーフローを起こす可能性のあるコードについては検出対象外である．以上の点において我々の検出手法は優位性があるといえる．

7. おわりに

本研究では，32 bit を超えるデータサイズで time_t 型を定義しているにもかかわらず，2038 年問題に起因する整数オーバーフローが起こるパターンの一部を検出する手法を示した．具体的には，ファイルシステムからの入出力時およびダウンキャスト時の整数オーバーフロー可能性を検出する手法を提案した．さらに，提案手法の有効性を示すため，検出手法に用いるツールを開発し，ツールを用いて GitHub 上の 874 の C 言語リポジトリを対象に検証した．検証の結果，我々のツールは 294 のプロジェクトで 3,463 の該当表現を検出し，32 bit を超える time_t 型環境においても 2038 年問題が脅威となることが示された．

今後の展望として，検出ツールの False Positive について提案手法を改善し，検出精度の向上を目指す．そのうえで，さらに多くのソースコードを対象に解析し，32 bit を超える time_t 型環境における 2038 年問題を含め，2038 年問題の影響範囲の広さを明らかにする．

参考文献

[1] Group, T.O.: POSIX.1 FAQ, The Open Group (online), available from ⟨https://www.opengroup.org/austin/papers/posix_faq.html⟩ (accessed 2023-12-01).

[2] Harshini, S., Kavyasri, K.R., Bhavishya, P., and Sethukkarasi, T.: Digital World Bug: Y2k38 an Integer OverflowThreat-Epoch, *JCSE International Journal of Computer Sciences and Engineering*, Vol.5, pp.106–109 (2017).

[3] 星名藍乃介, 穐山空道, 上原哲太郎：32 bit を超える time_t 型をもつ環境における 2038 年問題の検出手法の提案，マルチメディア，分散，協調とモバイルシンポジウム 2023 論文集，Vol.2023, pp.309–318 (2023).

[4] Microsoft: TIMEVAL (winsock.h) - Win32 app, Microsoft (online), available from ⟨https://learn.microsoft.com/ja-jp/windows/win32/api/winsock/ns-winsock-timeval⟩ (accessed 2023-12-01).

[5] IBM：UNIX，Linux，および Windows での標準データ・タイプ - IBM Documentation，IBM（オンライン），入手先 ⟨https://www.ibm.com/docs/ja/ibm-mq/9.1?topic=platforms-standard-data-types-unix-linux-windows⟩（参照 2023-12-01）．

[6] Project, T.N.: Announcing NetBSD 6.0, The NetBSD Project (online), available from ⟨https://www.netbsd.org/releases/formal-6/NetBSD-6.0.html⟩ (accessed 2023-12-01).

[7] Foundation, T.O.: OpenBSD 5.5, The OpenBSD Foundation (online), available from ⟨https://www.openbsd.org/55.html⟩ (accessed 2023-12-01).

[8] LKML.ORG: LKML: Arnd Bergmann: [GIT PULL] y2038: core, driver and file system changes, LKML.ORG (online), available from ⟨https://lkml.org/lkml/2020/1/29/355⟩ (accessed 2023-12-01).

[9] Microsoft: Time Management, Microsoft (online), available from ⟨https://learn.microsoft.com/en-us/cpp/c-runtime-library/time-management?redirectedfrom=MSDN&view=msvc-170⟩ (accessed 2023-12-01).

[10] 一般社団法人 JPCERT コーディネーションセンター：INT32-C. 符号付き整数演算がオーバーフローを引き起こさないことを保証する，一般社団法人 JPCERT コーディネーションセンター（オンライン），入手先 ⟨https://www.jpcert.or.jp/sc-rules/c-int32-c.html⟩（参照 2023-12-01）．

[11] Oracle：MySQL :: MySQL 8.0 リファレンスマニュアル :: 11.2.2 DATE，DATETIME，および TIMESTAMP 型，Oracle（オンライン），入手先 ⟨https://dev.mysql.com/doc/refman/8.0/ja/datetime.html⟩（参照 2023-12-01）．

[12] Microsoft: DATEDIFF (Transact-SQL) - SQL Server, Microsoft (online), available from ⟨https://learn.microsoft.com/ja-jp/sql/t-sql/functions/datediff-transact-sql?view=sql-server-ver16⟩ (accessed 2023-12-01).

[13] 日本規格協会：JISX3010:2003 プログラム言語 C (2003).

[14] CERT-C: INT02-C. Understand integer conversion rules - SEI CERT C Coding Standard - Confluence, CERT-C (online), available from ⟨https://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules⟩ (accessed 2023-12-01).

[15] 鈴木慶汰，窪田貴文，河野健二：ユーザレベルアプリケーションにおける 2038 年問題の検知と解析，技術報告 10，慶應義塾大学 (2019).
[16] Synopsys：Coverity 静的解析（SAST）ソフトウェア，Synopsys（オンライン），入手先 ⟨https://www.synopsys.com/ja-jp/software-integrity/security-testing/static-analysis-sast.html⟩（参照 2023-12-01）.
[17] manpages debian org: Debian Manpages, manpages debian org (online), available from ⟨https://manpages.debian.org/⟩ (accessed 2023-12-01).
[18] The Linux man-pages project: Maintaining Linux man-pages, The Linux man-pages project (online), available from ⟨https://www.kernel.org/doc/man-pages/maintaining.html⟩ (accessed 2023-12-01).
[19] jgamblin: jgamblin/Mirai-Source-Code, available from ⟨https://github.com/jgamblin/Mirai-Source-Code⟩.
[20] chenyahui: chenyahui/AnnotatedCode, available from ⟨https://github.com/chenyahui/AnnotatedCode⟩.
[21] bitnine oss: bitnine-oss/agensgraph, available from ⟨https://github.com/bitnine-oss/agensgraph⟩.
[22] kiddin9: kiddin9/openwrt-packages, available from ⟨https://github.com/kiddin9/openwrt-packages⟩.
[23] aergoio: aergoio/litetree, available from ⟨https://github.com/aergoio/litetree⟩.
[24] alexdobin: alexdobin/STAR, available from ⟨https://github.com/alexdobin/STAR⟩.
[25] jech: jech/polipo, available from ⟨https://github.com/jech/polipo⟩.

星名 藍乃介 （学生会員）

2023 年立命館大学情報理工学部卒業．現在，同大学大学院情報理工学研究科博士前期課程在学中．システムセキュリティ，ソフトウェア工学に興味を持つ．

穐山 空道 （正会員）

2010 年京都大学工学部情報学科卒業．2015 年東京大学大学院情報理工学系研究科創造情報学専攻修了．博士（情報理工学）．日本電信電話株式会社，産業技術総合研究所，東京大学を経て，2022 年 4 月より立命館大学情報理工学部セキュリティ・ネットワークコース准教授．メモリシステム，性能分析，仮想化技術等の研究に従事．

上原 哲太郎 （正会員）

1995 年京都大学大学院工学研究科博士後期課程研究指導認定退学．同大学院工学研究科助手，和歌山大学システム情報学センター講師，京都大学大学院工学研究科附属情報センター助教授，同大学学術情報メディアセンター准教授，総務省情報通信戦略局通信規格課標準化推進官を経て，2013 年より立命館大学情報理工学部教授．京都大学博士（工学）．デジタル・フォレンジック，システムセキュリティ等の研究に従事．共著に『基礎から学ぶデジタル・フォレンジック』（日科技連出版）等．