Vol.2019-OS-147 No.10
2019/7/25

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

鈴木慶汰[1]　窪田貴文[1]　河野健二[1]

概要：2038 年問題は，多くの既存のソフトウェアに深刻な損害を与える年問題の一種である．これは `time_t` の整数オーバーフローによって引き起こされる問題であり，開発者は，問題が実際に発生する 2038 年までにプログラムを修正する必要がある．2038 問題は既に広く知られているが，既存のツールではこれによって引き起こされるバグを検出することは難しい．これは，年問題はしばしばシステム特有の方法で時間表現をエンコードしており，このことを認識していないと検知することが難しいからである．UNIX ベースのオペレーティングシステムや C 標準ライブラリなどのシステムソフトウェアではこの問題は積極的に対処されているが，ユーザーレベルのアプリケーションに置けるバグは依然として未対処である．
本研究では，実際にアプリケーションを解析し，2038 年問題が本当に脅威になり得るということを示す．バグ検出を自動化するために，C 言語用の静的解析ツールを作成し，2012 年から 2018 年の 7 月 1 日から 7 月 10 日までの間に GitHub にアップロードされたすべての C 言語のプロジェクト (計 32,921）を解析した．解析を行った結果，コンパイルされたプロジェクトの 7.35 % に 2038 年問題のバグがあることがわかった．これには，不正確なログ出力などの軽いのバグから，Undefined Behavior を引き起こすような重大なバグが含まれていた．さらに一部のバグでは，プログラムの動作を変更したり，アプリケーションをクラッシュさせるような危険なバグが存在した．実際に発見されたバグのうち，Amazon Web Service Suuport tools や Linux Test Project などの大規模なプロジェクトを含む 5 つのプロジェクトで我々が作成したバグ修正パッチが確認／承認された．

1. はじめに

ソフトウェア開発において，年問題は非常に重要な問題である．これは時間表現を特殊な方法でエンコードすることによって引き起こされる問題で，既存のアプリケーションの多くはある日時を境に正常に動作しなくなる恐れがある．ほとんどの場合，年問題はハードウェア制約を克服するためのデファクトスタンダードによって引き起こされる．このため，問題のないプログラムを最初から作成するのは困難であり，開発者は時間の経過とともにソースコードを改良する必要がある．例として，2000 年問題は，マシンのストレージとメモリ不足によるものであった．このため，当時は年表現の最初の 2 桁を削除することがデファクトスタンダードであった．これにより，のちに 2000 年と 1900 年 (どちらも `00` として表される) を見分けることができなくなり，多くの開発者にソースコードを修正することを強いられた．

年問題は広く知られているが，それらを修正することは容易ではない．この原因の 1 つは，年問題によって引き起こされるバグは実際に問題が起きた後にのみ発生するからである．それまでは意図したとおりに動作するため，開発者はしばしばバグが存在することさえ気付かない．もう一つの原因は，既存のコンパイラの警告やバグ解析ツールでは年問題のバグを検知できないという点である．年問題を正しく検出するためには，どのようにして時間表現がエンコードされているかを理解する必要がある．しかし，既存のツールはこのセマンティクスを考慮していないことが多い．そのため，これらのツールではいくつかの一般的なバグは発見できるが，年問題そのものを発見するのには適さない．

年問題の一種として 2038 年問題がある．この問題は，UNIX ベースのオペレーティングシステムで発生し，`time_t` の整数オーバーフローに起因する．`time_t` は 1970 年 1 月 1 日からの経過時間を秒単位で保持していて，従来のシステムでは 32 bit 符号付き整数で定義されていた．しかし，この定義では 2038 年 1 月 19 日に経過秒数のオーバーフローが起きてしまう．これを防ぐために現在の多くのシステムでは 64 bit 符号付整数に再定義をしている．

[1] 慶應義塾大学
Keio Uniersity

ⓒ 2019 Information Processing Society of Japan

例えば，C 言語の標準ライブラリの一種である GNU C Library (glibc)[7] では，time_t の修正は 2000 年より始まり [1]，2019 年 1 月 31 日にリリースされたバージョン 2.29 で，すべてのアーキテクチャでのサポートが追加された [8]．time_t の定義が変更になったことによってそれまで 32 bit だと仮定していたアプリケーションが正常に動作しなくなってしまうため，対策が必要となる．

本論文では，実世界に存在するユーザレベルのアプリケーションを解析し，2038 年問題がアプリケーションの脅威になりうることを示す．具体的には，2038 年問題によって誘発されるキャスト関連のバグと I/O 関連のバグの 2 種類のバグを考える．どちらの種類のバグも現在のコンパイラや静的解析ツールでは見逃される可能性があり，実行中のプログラムの動作を暗黙で変更する，あるいは最悪の場合，終了させる可能性がある．

2038 年問題がユーザーレベルのアプリケーションで実際に問題があることを示すため，GitHub[3] に公開されているアプリケーションを用いて解析する．具体的には，2012 年から 2018 年の 7 月 1 日から 7 月 10 日の間に作成された全ての C 言語のプロジェクトを解析する．解析を行うために，簡単な静的解析ツールを作成した．

アップロードされていた 32,921 のプロジェクトのうち，3,973 をコンパイルすることができた．そのうち，7.35% に当たる 292 プロジェクトに 2038 年問題のバグが含まれていた．見つかったバグに対し，その特徴ごとに 5 種類に分類した．これらはフォーマットされた値をログなどに書き出すものや，値が直接キャストされてオーバフローを起こすようなものがあった．さらに，各バグをその重大度で分類した．この中には，プログラムの動作を変更，もしくは停止してしまうようなものも含まれていた．

発見したバグに対して，これらのバグを開発者に通知し，5 つのプロジェクトによってバグ修正パッチは承認された．これには，Amazon Web Service Support Tools [11] や Linux Test Project[6] などの大規模プロジェクトが含まれていた．パッチが承認されたその他のプロジェクトは gear-lib[17]，rfc5077[25] および FTL[22] であった．

本論文は次のように構成されている．2 章では，2038 年問題を説明し，ユーザーレベルアプリケーションが抱えるバグについて説明する．3 章では，解析を行うためのツールについて説明する．4 章では，それぞれのバグを分析し，2038 年問題が実際に脅威になりうることを示す．5 章では，関連研究を紹介する．

2. 2038 年問題

2.1 2038 年問題

2038 年問題は time_t の整数オーバーフローに関する年問題である．UNIX ベースのオペレーティングシステム (OS) では，1970 年 1 月 1 日からの経過時間 (UNIX EPOCH TIME と呼ばれる) を管理している．これは秒単位で time_t という型にエンコードされいる．従来のシステムでは time_t は 32 bit の符号付き整数として定義されていた．しかしこの定義では，2038 年 1 月 19 日に経過時間が 32 bit 符号付き整数の表現範囲を超えてしまい，整数オーバーフローが発生し，格納されていた値は負の数になってしまう．これを解消するため，多くのシステムでは現在は time_t は 64 bit の符号付き整数に再定義されている．

time_t の再定義によって，time_t を 32 bit と見なして作成されたプログラムを変更する必要がある．これは，各ソフトウェアレイヤによって異なる．次節では， 2038 年問題がソフトウェアスタックに及ぼす影響を説明し，実際に行われた修正を紹介する．さらに，ユーザーレベルのアプリケーションにおける問題を明確にし，既存のコンパイラやバグ解析ツールを使用してこれらのバグを検出することは困難であることを示す．

2.2 ソフトウェアスタックへの影響

このセクションでは，ソフトウェアスタックの各レイヤに存在する問題とそれらの修正を紹介する．具体的にはファイルシステムとオペレーティングシステム (OS) を紹介する．

1 つ目のレイヤはファイルシステムである．ファイルシステムでは，データへのアクセスのタイムスタンプをメタデータ内に保持している．これらのタイムスタンプは特定のビットサイズにハードコーディングされていることが多く，従来のファイルシステムでは，このビットサイズは time_t と同じサイズである．このため，これらのファイルシステムでは 2038 年以降のタイムスタンプを作成できない．これを修正するために，開発者達はファイルシステムの新しいバージョンを作成している．Ext4[2] では，オーバーフローを延期するためにタイムスタンプが 2 bit 追加され， 34 bit 符号付き整数に更新された [5]．

2038 年問題を抱えている他のソフトウェアレイヤとしてオペレーティングシステムである．OS では，time() や gettimeofday() のような time_t 関連のシステムコールは更新が必要になる．これらの関数は， time_t を 32 bit だと仮定して作られていた．これにより，正しい値を返さなくなってしまう．現在，Linux Kernel コミュニティは 2038 年問題を修正するために積極的にコードの更新を行っている [4]．

2.3 ユーザーレベルアプリケーションへの影響

ソフトウェアスタックの各レイヤと同様に，ユーザーレベルのアプリケーションも見直す必要がある．次の 2 節では，ユーザーレベルのアプリケーションにおける 2 種類の典型的な 2038 年問題のバグについて説明する．また，両

```
1  int cast_example(){
2      //  Use time to pick a random fortune.
3      i = (int)time(NULL) % numFortunes;
4      res = getStrByIndex(rsrc, i, &err);
5      if (U_FAILURE(err))
6          exit(-1);
7      return 0;
8  }
```

コード 1 Cast Bugged Program in moovweb/open-tritium

方のコンパイラまたは既存のデバッグツールで両方のバグを見つけるのが難しいことも説明し，2038 年問題のバグが危険であり，最悪の場合，プログラムを終了させる可能性がある理由について説明する．

2.3.1 キャスト関連のバグ (Cast Bug)

ユーザーレベルのアプリケーションにおける 2038 年問題のバグの 1 つはキャストバグ関連のバグ (Cast Bug) である．Cast Bug は `time_t` を小さいビットサイズの型にキャストすることによって引き起こされるものである．これは整数オーバーフローを引き起こす．

コード 1 はプロジェクト [18] で発見された Cast Bug の例である．このコードでは乱数を得るために `time_t` の値を使用している．3 行目で，関数 `time()` が呼び出されている．この関数の戻り値は `time_t`(64 bit) なので，`int`(32 bit) への明示的なキャストはバグとなる可能性がある．ここでは開発者は変数 `i` の範囲が 0 から `numFortunes` の間にあると仮定している．しかし，`time()` の戻り値が `int` の範囲を超えると，値は負の数として表現されてしまう．この場合，変数 `i` は 2038 年以降は負の数になり，これは開発者の意図しない動作である．また，この値は後に配列のインデックスとして使用されている (4 行目)．`i` が負の数の場合 index out-of-bounds となり， これによってプログラムが異常終了する．

このようなバグはほとんどのツールでは警告が出ない．これは，ツールは明示的なキャストを開発者の意図として解釈し，整数オーバーフローを引き起こす可能性があってもそれは開発者が意図していると想定するためである．gcc や clang のようなコンパイラの場合，最高の警告レベル (`-Wall`/`-Weverything`) であってもコード 1 に警告を出さない．また，KINT[26] では，配列のインデックスとして負の数を使うなど，いくつかの一般的なバグを検出することができるが，キャストを含む型変換は整数エラーと見なしていないため，2038 年問題固有のバグを検出するのは難しい．

2.3.2 I/O 関連のバグ (I/O Bug)

もう一種類のバグは I/O 関連のバグ (I/O Bug) である．I/O bug は `time_t` の値を I/O に書き込むものである．`time_t` の値が外部装置に出力されるとき，各プログラム

```
1  int create_backup(FILE *db) {
2      time_t d_t = time(NULL);
3      ...
4      fwrite(&d_t, 1, sizeof(time_t), db);
5  }
6  int restore_backup(FILE *db) {
7      time_t d_t;
8      ...
9      fread(&d_t, 1, sizeof(time_t), db);
10 }
```

コード 2 Example of I/OBug found in various projects

は `time_t` の読み／書きのビット幅に関して一貫していなければならない．書き込み時は `time_t` を 64 bit として実行しているのに，読み込み時は 32 bit として実行している場合，プログラムは値の最初の半分だけを読み取る可能性が出てしまう．これに対策するには，読み込みのプログラムと書き出しのプログラムがどちらも同じ `time_t` の定義の環境でコンパイルされている必要がある．

コード 2 は，さまざまなプロジェクトで見られる I/O Bug の典型例を示している [14], [15], [20], [21]．このコードは，最初の関数でバックアップファイルを作成し，それを 2 番目の関数で復元している．これは書かれた `time_t` が，読み出す時の `time_t` の型と一致している時はうまく動作するが，これらの型が矛盾していると問題が発生する．この例では，`fread()` で指定されたサイズが実際書き込まれたサイズと矛盾する場合，壊れたデータセットを読み出すことになりうる．

コード 2 は， `time_t` のビット幅が時間の経過とともに変化しなければ正しい．したがって，既存のコンパイラは警告を出さず，より一般的なバグをターゲットとしたデバッグツールでさえも危険とは見なさない．我々が作成したツールでは， `time_t` の値が外部デバイスに書き込まれたり外部デバイスから読み取られたりすると警告を出す．これらの警告は，開発者が読み書きプログラム間の一貫性をチェックすることを促す．

3. 2038 年問題のバグを検出するためのツール

ユーザーレベルアプリケーションにおける 2038 年問題のバグを検出するために単純なツールを作成した．このツールは 2038 年問題固有の問題を解析でき，簡単なデータフロー解析を行なっている．本ツールは LLVM コンパイラフレームワーク上で実装されており，LLVM-IR を解析する．

データフロー解析の必要性: 2038 年問題のバグを検出するには，`time_t` を保持した変数のデータフローを追跡する必要がある．我々のツールは前章で紹介した 1)Cast Bug(2.3.1) と 2)I/O Bug (2.3.2) に焦点を当てて解析を

行う．

一見すると，time_t 型の変数のキャストや書き出しさえチェックすれば十分のように思えるが，このようなチェックのみだと数多くの 2038 年問題のバグが未検出のままになってしまう．例えば，time_t の変数が一度 64 bit 整数型にキャストされ別の変数に格納されたのちに，その後さらに 32 bit 整数型にキャストされたとする．このような場合，2 番目のキャストは期待通りに動作しない可能性がある．time_t 値のデータフローを追跡することによって，我々のツールはより精密にバグを検知できるようになる．上記のように time_t 値が 64 bit にキャストされた後でも，我々のツールはデータフローを追跡を行うためである．

バグ検出: time_t 値は，以下のような gettimeofday() などの標準ライブラリ関数を呼び出すことによって取得される．これらの関数はデータフローの起点になるため *trigger functions* と呼ばれる．これらの関数の呼び出しが見つかったとき，返り値の time_t の値はデータフローグラフのルートノードと見なされる．本ツールでの *trigger functions* は time()，mktime()，および gettimeofday() を指定している．

time_t 値が fwrite() のような I/O ライブラリ関数に渡される場合我々のツールは警告を出力する．これは読み込み／書き出しの確認するビット幅を確認する必要があるためである．また，time_t の値のキャストがある場合も我々のツールは警告を出す．

最適化: 我々のツールは関数間の解析も行うため，trigger function の全てのデータフローを考慮する場合はスケールしない．この解析量を減らすため，我々は標準ライブラリの関数には 2038 年問題のバグはないと仮定し，ライブラリ関数に渡される time_t 値は正しく扱われていると，考える．この過程により，私たちのツールは標準ライブラリ内のデータフローは解析しない．これには localtime_r()，gmtime()，gmtime_r()，ctime()，ctime_r()，asctime()，asctime_r() そして srand() といった関数がある．

4. バグの解析

4.1 概要

この章では，実際にアプリケーションを解析し，2038 年問題が実際に脅威になりうることを示す．本解析のアプリケーションとして，2012 年から 2018 年の 7 月 1 日から 7 月 10 日までの間に作成された C ベースの GitHub プロジェクトを使用する．プロジェクトの取得には，ghtorrent[16] を使用する．

本解析は各プロジェクトの *Makefile* を使用し，ビルドプロセス中に解析を行う．したがって，解析は各 *Makefile* の設定に依存する．このため，各解析はモジュール単位での解析に限定される．また，各解析は最適化レベル (-O0) と

表 1 プロジェクト数

Category	Number
Uploaded	32,921
compiled	3,973
Bugged Projects	292
Bugged Project rate	7.35%
Warning Generated	2,043
Actual Bugs	1,739
Precision	84.91 %

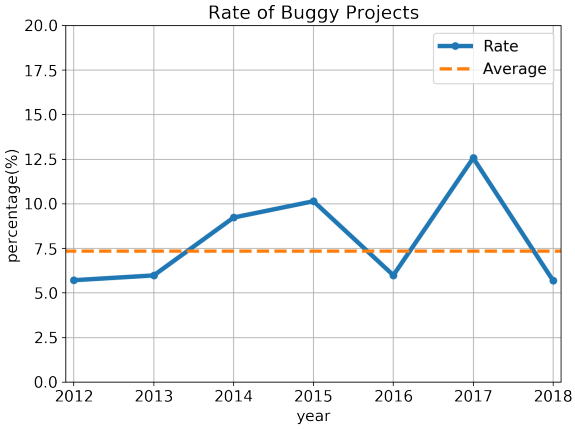

図 1 年単位のバグプロジェクトの割合

デバッグオプション (-g) を指定し，コンパイラを実行する．さらに，-Wformat 警告オプションと-Wshorten64-to-32 警告オプションを明示的に指定する．

本解析は 2 台のマシンで行われた．1 台は Intel Xeon X5675 と 16GB の RAM，もう 1 台は Intel Xeon X2620 と 64GB の RAM である．ツールは LLVM バージョン 8.0 で実装されていて，Ubuntu18.04.01(Linux4.15.0-39-generic) で動作している．

4.2 発見されたバグの数

表 1 はバグを含むプロジェクトの数とバグの警告の数を示している．GitHub にアップロードされていた 32,921 のプロジェクトのうち，我々の環境では 3,973 のプロジェクトをコンパイルできた．それらのうち，292 プロジェクトにバグが含まれていた．これはコンパイルできたプロジェクトの 7.35%であった．図 1 は，バグの含まれるプロジェクトの割合を年単位で示したものである．これからわかるように，バグを含むプロジェクトの割合は各年でおおよそ同じであるという点である．これは，開発者がソースコードに存在する 2038 年問題のバグに気付いていないことを示している．

我々のツールは全部で 2,030 の警告を出した．1 つのプロジェクトに複数のバグが含まれている可能性があるため，生成される警告数はプロジェクトの合計数と一致していない．生成された警告のうち，1,726 の警告が実際のバ

表 2　バグカテゴリ

カテゴリ名	定義
Print	printf() や fprintf() のように値をフォーマットしている I/O Bug
Direct Cast	time_t 値を直接キャストするバグ
arithmetic loss	演算結果の整数オーバーフローによって引き起こされるバグ
print diff	print と arithmetic loss の組み合わせ
raw write	write() のような関数で time_t 値を直接書き出す I/O Bug

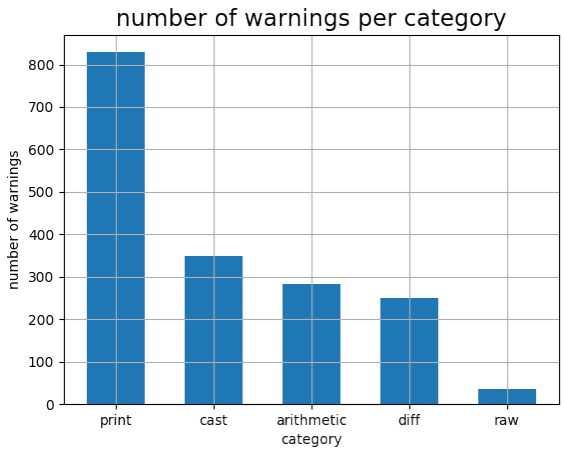

図 2　カテゴリごとに見つかったバグの数

グを指していた．よってツールの精度は 85.02%であった．これは私たちが使った非常に単純な方法でも十分効果的であることを示している．

4.3　バグ分類

見つかったバグの特徴をさらに調べるために，5 つのカテゴリに分類する．表 2 には各カテゴリとその説明を示す．詳しくは，4.3.1 節から 4.3.5 節で述べる．

図 2 は，カテゴリごとのバグの数を示す．最も発見されたバグは print であった．これは，ログや警告を生成する際に多くのアプリケーションでは time_t の値を使用するためである．2 番目に多かったバグは direct cast であった．これは主に，値が int など小さい型に格納された場合や，キャスト子の位置が間違っていた場合に発生した．これに続いて，arithmetic loss と print diff が続いた．両方とも，主に時間計算するときの状況で見つかった．5 つのカテゴリの中で最も少なかったのは raw write であった．これは，アプリケーションの多くは time_t をバイナリデータとして書き出すのではなく，正規化された値を出力しているためである．

4.3.1　print

Print bug は，printf() や fprintf() のような関数を使ってフォーマットされた time_t 値を書き出す I/O bug である．このカテゴリに含まれるバグでは， time_t を端末またはストレージに出力していた．ほとんどの場合，このバグは time_t がログや警告のタイムスタンプとして使われている状況で発見された．print bug は主に割り当てられたフォーマット識別子と引数の型の不一致が原因である．このため，ほとんどの print bug はフォーマット識別子を変更するだけで修正できる．

```
1 void LOG_MAIN(char * m){
2     printf("%u|%s", (uint)time(NULL), m);
3 }
```

コード 3　print bug の例

コード 3 は簡単な print bug の例である．このコードは printf() を使ってログを標準出力に出力している．3 行目に見られるように，time_t はタイムスタンプとして出力されている．しかし，この値は符号なしの整数に明示的にキャストされている．このため，time() の戻り値が uint の範囲を超えると，誤ったタイムスタンプが出力されてしまう．

4.3.2　Direct cast

Direct cast は time_t の値を小さいビットサイズの型に直接キャストするバグである．このバグは様々な状況で見られる．キャストバグの典型例としては time_t の値を int などの小さなビットサイズの型に格納するときの整数オーバーフローが含まれる．

Direct cast はプログラムの動作を変更する可能性が高いバグである．2 節で紹介したコード 1 は direct cast の一例である．前述のように，このバグは実行中のプログラムを終了させる可能性があるため危険なバグであると考えることができる．各バグの重大度については，4.5 節で詳しく述べる．

4.3.3　arithmetic loss

Arithmetic loss は direct cast の特殊なケースである．このカテゴリのバグは，主に算術演算によって引き起こされる整数オーバーフローである．ほとんどの場合この演算は減算である．演算結果はより小さなビットサイズ型に格納されることによって，整数オーバーフローが発生する．ほとんどの arithmetic loss は単に difftime() のような関数を使うことで解決することができる．

Code4 は，pi-holes/FTL[22] で発見された arithmetic loss の例である．このコードは条件節で time_t の値を使用している．6 行目にあるように，time() と time()%INTERVAL の結果は変数 lastRun に格納されている．しかし，time()%INTERVAL の結果は 0 から INTERVAL の範囲にあり最大でも 3600 であるため，引き算結果は int の範囲を超えてしまう可能性がある．これにより正しくない値が格納されてしまう．さらにこの変数は，後に条件節で使用される (9 行目)．変数は正しい値を保持していないため，条件

```
1 #define INTERVAL 3600
2 int lastRun = 0;
3 void *GC_thread(void *val){
4     ...
5     lastRun = time(NULL) - time(NULL) %
          INTERVAL;
6     if(time(NULL) - lastRun >= INTERVAL)
7         ...
8 }
```

コード 4 arithmetic loss の例 (pi-holes/FTL)

は意図したとおりには機能せず，プログラムの動作が変更されてしまう．

4.3.4 Print diff

Print diff は print と arithmetic loss の組み合わせである．Print と同様，このバグは主にログを作成するときに出現する．したがってその特徴は Print と同様となっている．

4.3.5 Raw Write

Raw write は，I/O bug の特殊なケースである．このバグはバックアップファイル等を作成するときに出現し，`write()` や `fwrite()` などの関数を使用し，`time_t` を直接ファイルに書き込む．見つかったバグはすべてストレージへの書き込みで，`send()` または `sendto()` への呼び出しは含まれていなかった．これは，ネットワークアプリケーションの開発者は `time_t` がそれぞれのマシン環境によって異なることを考慮した上でコードを書いていると考えられる．Raw write の例としては，2.3.2 節のコード 2 がある．

Raw write は見つけるだけでなく，修正するのも困難である．これは，迂闊に修正をしてしまうと既存のデータを破壊してしまう可能性があるからである．`time_t` のビット幅が変更されると， ビット幅の不一致が原因でデータが破損してしまう．これを防ぐには，ビット幅をハードコーディングしてデータを以前の 32-bit データ構造に変更するか，32 bit で保存された古いデータは 64 bit に変換して新しいデータ構造に変換する必要がある．前者は，時間表現は依然として 2038 年問題を抱えるが，データは読み込むことができる．後者は，各データを `time_t` の新しいエンコーディングに変換する必要がある．実際にどの方法を使用するかを選択するのは各開発者の責任であるが，全体としてコードと 2038 年問題自体を深く理解する必要がある．

4.4 明示的/暗黙的キャストの数

Direct cast や arithmetic loss などの cast bug には，明示的キャストと暗黙的キャストの両方が含まれていた．表 3 は，各カテゴリで見つかった数を示している．また、print もフォーマット記述子と値の不一致によってキャストが起きる可能性があるため考慮する．

表 3 暗黙的/明示的キャスト

category	explicit cast	implicit cast
print	366	97
direct cast	213	137
arithmetic loss	101	183

```
1 time_ms getTimeStamp() {
2   return (uint32_t)time(NULL);
3 }
```

コード 5 unsigned int への意図的なキャスト

表 4 カテゴリごとの重大度

category	LV.0	LV.1	LV.2
print	2	42	785
direct cast	1	220	129
arithmetic loss	0	64	220
print diff	0	0	241
raw write	0	35	0
合計	3	361	1375

いくつかの明示的なキャストは `time_t` が 32 bit の時は正しいとみなせるが，ビット幅が変更されることによってバグになるケースが含まれていた．その中でも一部のキャストは，2038 年問題対策として行われていた．これは `time_t` の値が，負の値にならないように 32 ビットの符号なし整数に明示的にキャストされていた．コード 5 は，さまざまなプロジェクトで見られた、意図的な明示的キャストの簡単な例を示している [10], [23]．関数 `gettimestamp()` は `time()` をラップして戻り値を `uint32_t` にキャストしている．これにより，戻り値が負の数に折り返されるのを防ぐことができる．このようなケースは明示的なキャストであるため，ツールは警告を出さない．したがって，これらを既存ツールを用いて見つけるのは困難である．この他にもコード 3 もこれに当てはまる．

4.5 各バグの重大度

2038 年問題のバグの中には検出が難しいだけでなく危険なものもある．これを示すため，各バグを 3 つのレベルに分類する．表 4 はバグカテゴリ別に各レベルの結果を示している．各レベルの定義は次のとおりである．

- LV.0 プログラムを終了させる可能性が高い
- LV.1 プログラムの動作を変更する可能性が高いが，動作を終了する可能性は低い
- LV.2 プログラム自体には害を及ぼさない

見つかったバグの半分以上が LV.2 に分類された．それらの大部分は print や print diff といったログ関連のバグであった．これらはログに表示された値は正しくないが，プログラムの動作には影響しない可能性が高い．そのた

```
1 void make_file(){
2     char fname_1[1024];
3     sprintf(fname_1,
4         "bu.%d.%d", (int)time(NULL), pid);
5     ...
6 }
```

コード 6　LV.1 の print bug

```
1 void lv0_bugs(){
2     char * tm[8];
3     ...
4     sprintf(tm,"%lu",
5             (long)(time(0)/100000));
6 }
```

コード 7　LV.0 の print bug

め，これらを LV.2 バグと見なす．

より注意が必要なのは，LV.0 と LV.1 に分類されたバグである．どちらのカテゴリもバグは実行中のプログラムに深刻なダメージを与える可能性があるため，同程度に危険である．すでに紹介したコードもこれらに属するバグである (コード 2,4)．

注目すべき点は，print も重大なバグを抱えているという点である．コード 6 はプロジェクト [12] で見つかった LV.1 の print bug である．このコードではファイル名に `time_t` を使用する．これを生成するために `sprintf()` を用いている (10 行目)．しかし，その過程で `time_t` の値は明示的に `int` にキャストされいる．これは 2038 年以降すでに存在するファイル名が作成される可能性があり，もしファイル名が存在する場合，ファイルの内容は上書きされてしまう．これにより，プログラムが古いデータを期待しているにも関わらず新しいデータを読み込んでしまうとプログラムの動作を変更されてしまう．

LV.0 の print bug は 6 のコードとよく似ているが，実行中のプログラムを終了させる可能性がある．コード 7 は 2 つのプロジェクト [24], [28] で発見された例である．Code6 のように，`time_t` は `sprintf()` を使って文字列に書き込まれる．しかし，文字列は 8 バイト配列として宣言されているため `time_t` の値がこの範囲を超えると， undefined behavior が引き起こされてしまう [27]．

4.6　発見されたバグの修正

一部の見つかったバグに関して，実際にバグ修正のパッチを作成し，GitHub 経由で開発者に送った．我々の修正パッチは 5 つのプロジェクトで承認された．表 5 にこれらのプロジェクト一覧を示す．これには，Linux Test Project[6]，pi-holes FTL[22]，AWS Support tools[11] などの大規模プロジェクトが含まれている．

表 5　修正が承認されたプロジェクト

Project Name	Stars	Contributors	Forked
AWS Support tools	325	19	205
FTL	303	12	55
gear-lib	1510	6	414
Linux Test Project	981	169	519
rfc5077	198	10	43

```
1 - int get_boot_time() {
2 + time_t get_boot_time() {
3     int uptime = (int)upsecs;
4     ...
5   return time(NULL) - uptime;
6 }
```

コード 8　AWS Support Tools の arithmetic loss bug

AWS Support tools[11] には，direct cast bug と arithmetic loss bug の 2 つのバグがあった．AWS Support tools は，Amazon が提供するクラウドサービスである Amazon Web Service[9] 提供および使用されるツールである．このコードにバグが含まれていると，サービスを使用している全てのアプリケーションが影響を受ける可能性がある．

Code8 は AWS Support Tools で発見された arithmetic loss の簡易版のコードである．このコードでは，アプリケーションの現在のランタイムを返す関数を定義している．もともと，`get_boot_time()` の戻り値は `int` であった．しかし，戻り値が `int` の範囲を超えると，この関数は意図しない負の値を返す．これを避けるためには，関数の戻り値も `time_t` として宣言するのが良い．このバグを含むバグ修正パッチは開発者によって承認され，すでにマージされている．

4.7　False Positive

今回の解析では我々のツール用いて，様々なバグを検知できた．しかし，我々のツールは依然としていくつかの false positive が存在した．これは，我々のツールはそれぞれのコードの意味そのものは考慮していないためである．これによって開発者が実際に意図していた場合でも警告を出してしまう．例えば明示的なキャストが実際に開発者の意図であった場合でも，我々のアプローチではキャスト自体の意味は考慮しないため警告を出す．また，演算処理が正しく行われていて結果が実際に正しい型に格納されている場合でも，演算結果が確実に格納する型の範囲を超えないかどうかは考慮していないため，警告が出てしまう．

5.　関連研究

整数エラーの検出:　Cast Bug の中には，一般に整数エラーと考えることができるものがあり，これらのバグはを

検出するために多くの既存のツールが研究されている．

KINT[26] では整数エラーを検出するために静的解析を使用している．正確に整数エラーを検出するために，KINT では不変条件とユーザー定義の条件を使用している．KINT はこれらの条件を破る可能性のある整数エラーを検出できるが，キャストを含む変換そのものは整数エラーとは見なさない．したがって，いくつかのキャストバグは検出されないままとなってしまう．

PREFIX+Z3[19] でも整数エラーを検出することができる．しかし，PREFIX+Z3 では，誤検出を避けるために明示的なキャストを含む警告はスキップされる．これは今回のケースのように明示的なキャストも問題になる可能性があるため，問題となってしまう．このため，このツールで 2038 年問題のバグをすべて検出することは容易ではない．また，I/O Bug のようなバグはいずれのツールでも発見するのは難しい．

Undefined Behavior Bugs: 重大な 2038 年問題のいくつかは Undefined Behavior であった．Wang et al. [27] は undefined Behavior によって， コンパイラがプログラムの動作の一部を変更することを示している．STACK は，ユーザーが自分のコードを完全に理解し全ての動作がユーザの意図であり， Undefined Behavior は引き起こさないという C 言語の仕様を利用してコードを解析する．これにより STACK は undefined Behavior バグを検出するのに適しているが，2038 年問題のすべてのバグが undefined behavior を引き起こすわけではないため STACK を使用するのは難しい．

年問題の検出*:* 一部のツールは，年問題のバグを検出するために作成されている．最近問題となった年問題として 2000 年問題 (Y2K 問題) がある．

AnnoDomoni[13] は，COBOL ソースコード中の Y2K 問題のバグを検出するツールである．当時, 多くの COBOL プログラムが，Y2K 問題の影響を受けていた．AnnoDomoni は，開発者が指定したアノテーションを使用して，Y2K 問題のバグを検出する．次にこのツールは型チェックによってソースを分析し，警告を発する．最後に開発者の注釈に従ってソースも変更される．私たちのターゲットは C 言語のプログラムであるため，AnnoDomoni は使用できない．

6. まとめ

本論文では，2038 年問題は多くの既存のアプリケーションにとって脅威となる可能性があり，対応が必要であることを示した．また，2038 年問題のバグは最先端のツールやコンパイラでは検出が難しいことを示した．これらを示すために，GitHub にアップロードされた既存のアプリケーションを解析した．この解析により，問題はまだ多くのプロジェクトにあり，ほとんどのバグは何年にもわたって修正されないままであることがわかった．また，バグのあるプロジェクトの約半分が 2038 年以降に動作が変更される可能性が高いことも示した．

謝辞 本研究は，JST，CREST，JPMJCR19F3 の支援を受けたものである．

参考文献

[1] : The GNU C Library: Modify time_t to long int (2000).
[2] : ext4 Data Structures and Algorithms (2018).
[3] : GitHub (2018).
[4] : Linux Newbies The year 2038 problem (2018).
[5] : Linux Newbies The year 2038 problem vfs (2018).
[6] : linux-test-project (2018).
[7] : The GNU C Library (2018).
[8] : The GNU C Library: Y2038: Add 64-bit time for all architectures (2018).
[9] : Amazon Web Services (2019).
[10] ADiamondPlum: Onboard-SDK (2016).
[11] awslabs: aws support tools (2018).
[12] davidyuan1989: 15213 (2014).
[13] Eidorff, P. H., Henglein, F., Mossin, C., Niss, H., Sørensen, M. H. and Tofte, M.: AnnoDomini: From Type Theory to Year 2000 Conversion Tool, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, New York, NY, USA, ACM, pp. 1–14 (online), DOI: 10.1145/292540.292543 (1999).
[14] else: mosquitto (2014).
[15] emutex: mosquitto (2016).
[16] Gousios, G.: The GHTorrent dataset and tool suite, *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, Piscataway, NJ, USA, IEEE Press, pp. 233–236 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2487085.2487132⟩ (2013).
[17] gozfree: libraries (2016).
[18] moovweb: open-tritium (2014).
[19] Moy, Y., Bjorner, N. and Sielaff, D.: Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis, Technical Report MSR-TR-2009-57 (2009).
[20] pbingogh: mosquitto-optimized (2016).
[21] peppo65: mosquitto (2016).
[22] pi-holes: FTL (2018).
[23] qorost: Onboard-SDK (2016).
[24] thawkins: skipfish-usap (2015).
[25] vincentbernat: rfc5077 (2018).
[26] Wang, X., Chen, H., Jia, Z., Zeldovich, N. and Kaashoek, M. F.: Improving Integer Security for Systems with KINT, *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA, USENIX, pp. 163–177 (online), available from ⟨https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang⟩ (2012).
[27] Wang, X., Zeldovich, N., Kaashoek, M. F. and Solar-Lezama, A.: Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, ACM, pp. 260–275 (online), DOI: 10.1145/2517349.2522728 (2013).
[28] zhoudaqing: skipfish (2017).

ⓒ 2019 Information Processing Society of Japan