静的検査ツールにおける警告箇所の版間追跡による確認コスト削減手法

渥美 紀寿[1,a)]　桑原 寛明[2,b)]

概要：コーディングにおける誤りを早期に発見するため，様々な静的検査ツールが提案され，ツールとして実現されている．しかし，膨大な数の警告や，false-positive の割合が大きいことなどの理由により，開発者は静的検査ツールの利用に消極的である．本研究では，静的検査ツールの警告箇所に対し，開発者が問題ないと判断した箇所を版間で追跡し，以降の検査における警告から除外することによって確認すべき警告箇所を削減する手法を提案する．

キーワード：静的検査ツール，プログラム解析，保守支援

A Version Tracking based Cost Reduction for Screening Alerts of Static Check Tools

Noritoshi Atsumi[1,a)]　Hiroaki Kuwabara[2,b)]

Abstract: Various static check tools are proposed and implemented to detect coding errors at early stage of development process. However, many developers do not use these tools aggressively because these tools report too many alerts including high rate of false-positive. In this paper, we propose a method to reduce alerts that developers should check to be defects or not. Our method uses version tracking to exclude the alerts that developers have decided to be not defects in the past versions.

Keywords: Static Check Tool, Program Analysis, Software Maintenance

1. はじめに

ソースコードの品質向上を目的として，コーディング規約の制定，静的検査ツールを用いた検査，コードレビューなどが実施される．

コーディング規約とはソースコードの保守性や再利用性，信頼性の向上を目的とし，ソースコードを記述する際に守るべきルールであり，ソフトウェアの開発プロジェクトごとに定められている．コーディング規約には K&R スタイル [1] や GNU コーディングスタンダード [2] などのコーディングスタイルに関する規約や，MISRA-C[3] や CERT コーディングスタンダード [4] などの不具合を埋め込まないための規約がある．

静的検査ツールには，コーディング規約に準拠していることを確認するための QAC[5] や CX-Checker[6] などのツールの他，不具合を検出するための Splint [7]，Cppcheck [8]，Clang Source Code Analyzer [9]，PMD[10]，Checkstyle [11] など様々なツールがある．

コードレビューでは，コーディング規約に準拠しているか，コードが複雑過ぎないか，不具合を引き起こさないかなどについて人手により目視検査する．既存の静的検査ツールは万能ではなく，検出できない不具合も存在するため，

[1] 名古屋大学
Nagoya University, Furo-cho, Chikusa-ku, Nagoya 464–8601, Japan
[2] 立命館大学
Ritsumeikan Uniersity, 1-1-1 Noji-higashi, Kusatsu, Shiga 525-8577, Japan
[a)] atsumi@nagoya-u.jp
[b)] kuwabara@cs.ritsumei.ac.jp

```
  ...
  if ((err = SSLHashSHA1.update(&hashCtx,
                                &serverRandom)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx,
                                &signedParams)) != 0)
      goto fail;
      goto fail;
  if ((err = SSLHashSHA1.final(&hashCtx,
                               &hashOut)) != 0)
      goto fail;
  ...
fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
  return err;
```

図 1: iOS 7.0.6 より前のバージョンの脆弱性コード

コードレビューによる検査が必要となる．しかし，静的検査ツールによる検査を行うことによって，静的検査ツールでは検査できない問題を重点的にコードレビューで実施することができ，コードレビューのコストを低減することが可能である．

iOS 7.0.6 で修正された不具合 [12] (問題のコード断片を図 1 に示す) のように静的検査やコンパイラの警告によって検出される単純な問題が製品のリリースバージョンにおいて発生している．このような不具合を防ぐために，静的検査ツールを利用した検査は有用である．

しかし，開発現場において静的検査ツールはあまり利用されていないのが現状である．Johnson らの調査 [13] では，静的検査ツールが利用されない原因として下記のことが挙げられている．

(1) false-positive が非常に多い
(2) 警告の数が多過ぎて処理できない
(3) プロジェクトに適した検査をするようにカスタマイズすることが困難
(4) チームで検査項目等の設定を共有することが困難
(5) 警告メッセージの内容が理解できない
(6) コーディング作業にあったツール統合ができない

これらの問題点を解決するために，false-positive を減らすための手法 [14], [15] や警告を重要な順にランキングするための手法 [16], [17] などの研究が行われている．しかし，これらの研究では，開発者が警告を無視してしまう問題を考慮していない．

本研究では，静的解析ツールの精度に関する問題ではなく，検出された警告を開発者が効率良く確認するための手法を提案する．これを実現するために，過去のバージョンにおいて，一度確認した警告を検査対象のバージョンの警告から除外する手法を提案する．既存の静的検査ツールでは，警告箇所に対して版を跨いだ支援をしておらず，開発者は修正のたびに変更箇所のみに着目して問題の有無を確認しなければならない．提案手法ではこの問題を解決する．

2. 静的検査ツール

静的検査ツールは非常に多く開発されており，それぞれのツールには解析方法や検出可能な欠陥の種類，検出精度に違いがある．パターンマッチによる検査ツールとして，RATS[18] や ITS4[19]，Flawfinder[20] などがある．これらのツールはバッファーオーバーフローやレースコンディションなどで，不具合を引き起こす可能性の高い関数の使い方をパターンとして記述し，それらを脆弱性データベースとして記録し，そのパターンにマッチするソースコード中の記述を検出する．false-positive は非常に多いが，高速に検査することが可能である．構文解析による検査には Splint[7] や Cppcheck[8]，Clang Static Analyzer [9] などがある．これらのツールはバッファオーバーフローやメモリリーク，不正な NULL ポインタ参照，インタフェースとの不整合など，様々な欠陥を検出することが可能である．

Chatzieleftheriou らの調査 [21] では オープンソースの静的検査ツール 4 つ (Splint，UNO，Cppcheck，Frama-C) と商用のツール 2 つ (Parasoft C++ Test，Com. B) に対してそれぞれのツールの欠陥検出能力を調査しており，それぞれのツールで検出可能な欠陥が異なること，およびその精度が異なることを示している．また，Zitser らの調査 [22] でも同様にオープンソースの静的検査ツール 4 つ (ARCHER，BOON，Splint，UNO) と商用ツール (PolySpace C Verifier) を対象に評価を行っており，検出可能な欠陥の違いおよび検出精度について議論している．

その他のツールについても同様に検出可能な欠陥や検出精度にはそれぞれ違いがある．そのため，複数の静的検査ツールを組み合わせることでより多くの欠陥を検出でき，ソースコードの品質を向上させることが可能となる．しかし，Johnson らの調査 [13] で挙げられている問題点のうち，

(1) false-positive が非常に多い
(2) 警告の数が多過ぎて処理できない

の 2 つの問題点をさらに悪化させることになる．本研究ではこれらの問題点を解決するために，未確認の警告箇所を全て提示しつつ，開発者が確認すべき警告箇所を削減するための手法を提案する．

3. 関連研究

静的検査ツールを用いたソースコードの品質向上のために，false-positive の削減や false-negative の削減などの精度向上を目的とした研究，重要な警告を優先して修正することを目的とした警告の提示方法に関する研究，静的検査ツールの利用を促進するための研究が行われている．本章ではこれらの研究について紹介する．

3.1 静的検査ツールの精度向上

Thung らは，FindBugs，PMD，Jlint の 3 種類の静的解析ツールを対象に，false-negative について評価している [14]．false-negative とは，本来欠陥であるはずだが検出されないものを指す．彼らは，3 種類の OSS を対象に実験を行い，FindBugs と PMD は false-negative の抑制に比較的有用であると述べている．しかし，いずれのツールの評価でも誤検出を防ぐことはできず，静的解析ツールに挙げられている問題は解決できていない．

Nanda らは，NULL ポインタの間接参照に関して，path-sensitive で context-sensitive なメソッドを跨いだ解析をすることで，false-positive を削減する手法を提案している [15]．彼らは 3 つの商用製品に対して FindBugs，Jlint と比較し，彼らの手法が他の静的検査ツールより多くのバグを検出可能であることを示した．また，他のツールより false-positive が少ないことを示している．

3.2 静的検査ツールの警告提示法

Muske らは，誤った警告を分割する手法を提案した [23]．最初に警告の等価性によってクラスに分割し，それぞれのクラスの 1 つが誤った警告であれば，そのクラス全てを誤ったクラスとして判定する．次に警告が指すコード中で参照される変数の変更箇所の類似性に基づいてクラスを分割する．彼らの実験では，これらの分割により，警告の 60 % は冗長な警告であることが示されている．

Shen らは，false-positive を抑制するために，FindBugs の警告に対し，true-positive の警告を上位に提示するための 2 段階のランキング手法を提案している [16]．第一段階ではあらかじめバグパターンごとに定められた欠陥の可能性によってランキングする．第二段階ではユーザからのフィードバックにより，ランキングを最適化する．彼らは 3 つの Java プロジェクト (AspectJ，Tomcat，Axis) に適用し，上位に提示される警告がオリジナルの FindBugs と比べて適合率，再現率，F1 値が向上したことを示している．

3.3 静的検査ツール利用促進

Tripp らは，静的検査ツールが生成する警告の一部をユーザにフィードバックさせ，そのフィードバックに基づいて統計的学習により，警告を分析する手法を提案している [24]．彼らはこれを実現するためのツール ALETHEIA を開発した．また，彼らの実験では，200 の警告をユーザに分類させることによって，適合率を挙げることができている．

新井らは，静的解析ツールを利用する開発者の欠陥修正に対するモチベーションを向上させ，報告された欠陥がより多く修正されるような仕組みを提案している [25]．彼らは提案手法を導入したツールを開発し，被験者実験を行ったところ，ツールを利用することによっておよそ 1.5 倍ほど多くの欠陥が修正されることを確認している．

4. 版間追跡による確認コストの削減

多くの静的検査ツールは，問題のあるソースコードのファイル名，行番号，問題の説明を警告として出力する．開発者はこの説明およびソースコードを参照し，不具合に繋がる場合には修正を，そうでなければ無視する．開発者はコーディング中に静的検査ツールを用いて随時ソースコードを検査することでソースコードの品質を確保することができる．ソースコードを修正するたびに静的検査ツールによる検査を行うことによってソースコードの品質を確保することが可能である．しかし，既存の静的検査ツールは警告箇所に対して版を跨いだ支援をしていない．そのため，検査をするたびにソースコードの修正箇所に着目して問題の有無を確認しなければならない．

本研究では，過去のバージョンにおいて静的検査ツールの警告箇所を確認し，問題ないと判断した箇所を除外して開発者に結果を提示することで，警告箇所の確認コストを削減する手法を提案する．過去のバージョンで問題ないと判断した箇所を現バージョンで除外できるようにする方法として，除外したい箇所をコメント等でマークする方法がある．しかし，この方法ではその箇所を修正した際に別の問題が発生する可能性があり，除外して良いかどうかを修正時に再度判断する必要がある．また，コメントの修正は忘れがちであり，未確認の問題が隠蔽される恐れがある．本研究では，バージョン間のソースコードの行単位での対応関係を利用し，現バージョンの警告行に対応する前バージョンの警告内容が現バージョンで出された警告内容と同じである場合，それを確認済みの警告とみなす．図 2 に提案手法の概要を示す．

開発者は静的検査ツールの警告箇所を確認し，問題ないと判断した場合，それを判断したバージョン，ファイル名，行番号，警告内容を記録する．ソースコードを編集後，再度静的検査ツールを実行した際には，ツールが生成した結果から，それまでに問題ないと判断した警告箇所と一致する箇所を除外し，未確認の警告箇所のみを提示する．このようにすることによって，開発者はそれまでに確認した警告箇所を再度確認する必要がなく，変更された箇所のみを確認することができる．

4.1 版間の追跡

Git や Subversion などの版管理ツールにはファイル中のそれぞれの行をどのリビジョンで誰が最後に変更したかを示す blame 機能がある．図 3 は git を用いた blame 機能を示している．

図 3 では，リビジョン cc763699 の test1.c がリビジョン d0efe358 で test1.c と test2.c に分割され，リビジョン 6c9eff05 で test1.c の関数 f が変更された場合を示してい

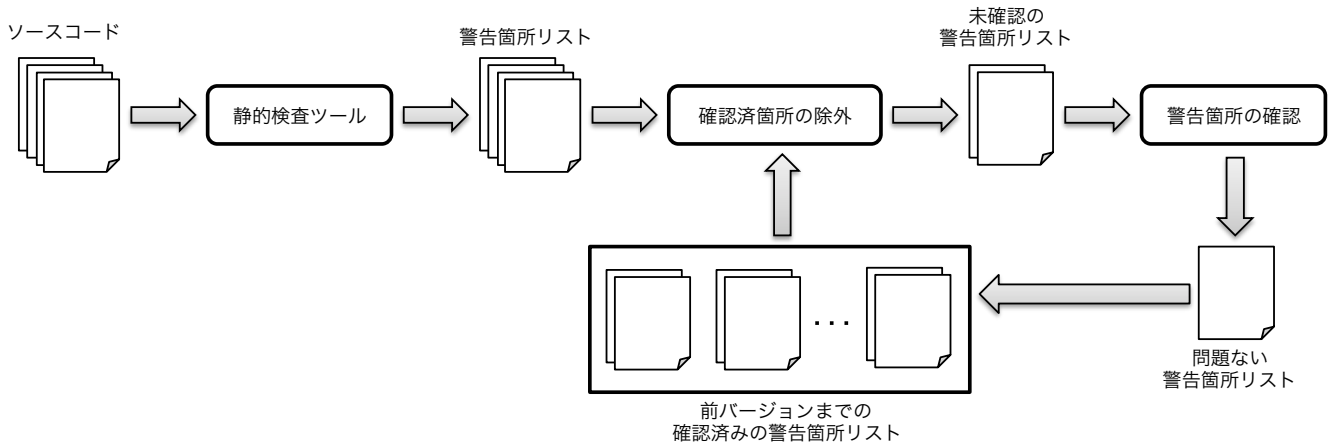


図 2: 提案手法の概要

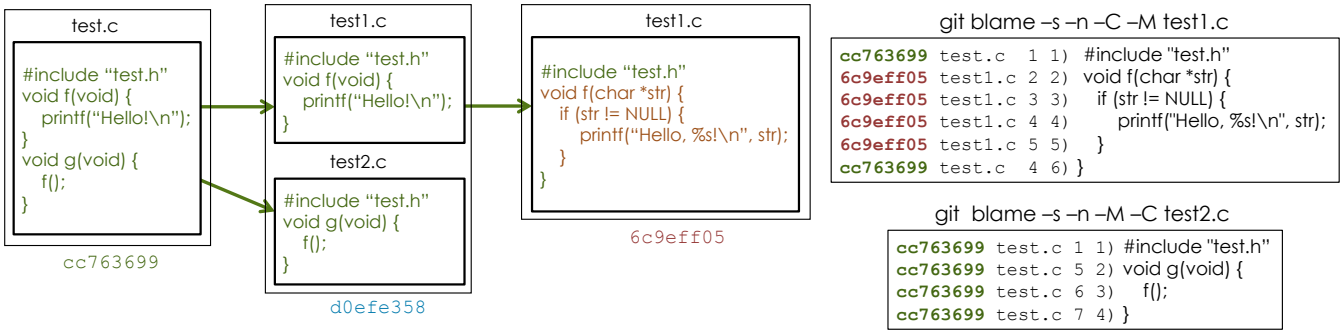


図 3: git-blame による変更行の追跡

る．リビジョン 6c9eff05 では test2.c を変更していない．git blame の結果には，最後に変更されたリビジョン，ファイル名，行番号，現在の行番号，該当行が示される．リビジョン 6c9eff05 で git blame -s -n -M -C test1.c を実行すると，1 行目は最初のリビジョン cc763699 から変更がないこと，2 行目から 5 行目はリビジョン 6c9eff05 で変更されたこと，6 行目は リビジョン cc763699 から変更がないことがわかる．同様に git blame -s -n -M -C test2.c を実行すると，1 行目から 4 行目までの全てが最初のリビジョン cc763699 から変更がなく，それぞれ対応する行が 1, 5, 6, 7 であることがわかる．

git blame のオプション -M はファイル内での行の移動やコピーを検知し，-C はファイルを跨った移動やコピーを検知する．そのため，図 3 のリビジョン cc763699 から d0efe358 の変更のようにファイルを分割した場合やファイル名の変更などにも対応して追跡することが可能である．

本研究では版管理ツールのこの機能を利用し，未確認の警告箇所のみを提示することによって，開発者の確認作業のコストを削減する．

4.2 確認済み箇所の記録

図 4 に Splint と Cppcheck による検査結果を示す．このように静的検査ツールの出力結果はツールごとにそれぞれ異なっているが，多くのツールでは，ファイル名，行番号，警告内容が示される．

```
test1.c: (in function f)
test1.c:8:5: Return value (type char *) ignored:
  fgets(input, 10,...
  Result returned by function call is not used.
  If this is intended, can cast
  result to (void) to eliminate message.
  (Use -retvalother to inhibit warning)
```

(a) Splint

```
[test1.c:8]: (error) Buffer is accessed
    out of bounds.
```

(b) cppcheck

図 4: 静的検査ツールの出力結果

提案手法において，開発者が問題ないと判断した場合，その判断をしたリビジョン番号と，ファイル名，行番号，警告内容の他に，版管理ツールの blame 機能を用いて，該当行を最後に編集したリビジョン番号と，ファイル名，行番号を記録する．この時記録する警告内容は，確認済み箇所の除外において同種類の警告であることを判定するた

めに利用する．そのため，静的検査ツールが生成する警告メッセージそのものではなく，警告の種類を識別するためのツールが出力する ID やメッセージの一部を警告内容として記録する．

4.3 確認済み箇所の除外

静的検査ツールを実行した際に生成される警告には，下記の異なる意味を持つ警告が存在するが，既存の静的検査ツールではこれらを区別することができない．

(1) 過去に確認し，修正する必要がないと判断した警告
(2) 過去に確認し，修正すると判断したが未修正の警告
(3) 過去にも警告されていたが，確認していない警告
(4) 過去のバージョンではなかったが，新しく出た警告

(1) の警告がソースコードを修正し，検査するたびに生成されると，開発者が上記区別をする必要があり，問題点を確認するコストがかかる．そのため，多くの場合，ソースコード中の変更箇所を中心に確認するが，静的検査ツールがファイルを跨った問題を検出する場合に，その警告を見逃してしまう．そのため，これらを適切に区別することが必要である．本研究では，(1) の警告を除外することで，開発者が問題を確認するコストを低減する．

開発者が静的検査ツールを実行し，過去のバージョンにおいて確認済みの記録がある場合には，その情報を用いて警告箇所から除外した情報を開発者に提示する．具体的には，検出された警告箇所の全行に対して，版管理ツールの blame 機能を用いて最後に編集されたリビジョン番号，ファイル名，行番号を調査し，それが確認済み箇所と一致し，かつ警告内容が一致した場合にはその警告を除外する．

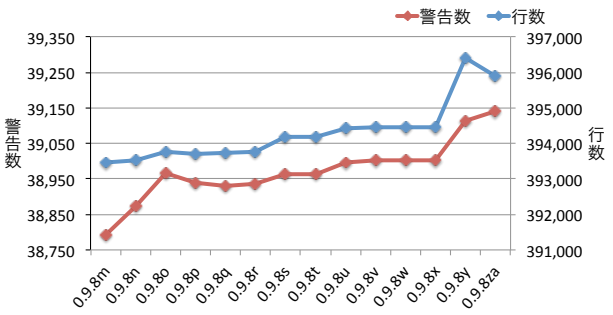
5. OSS を対象とした調査

本稿では，(1) 既存のソフトウェアにおいて静的検査ツールによる警告数がバージョンの進化に伴なってどのように変化しているか，(2) 提案手法によりどの程度警告箇所を削減できるかについて調査した．対象としたソフトウェアとバージョンを以下に示す．使用した静的検査ツールは Splint 3.1.2 である．

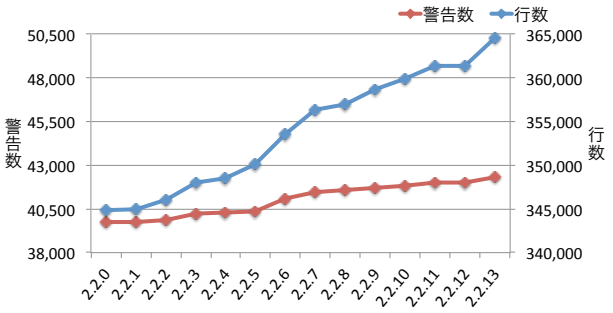
- OpenSSL (0.9.8m ～ 0.9.8za)
- Dovecot (2.2.0 ～ 2.2.13)
- Courier-IMAP (4.9.1 ～ 4.15)
- Sendmail (8.14.0 ～ 8.14.9)

5.1 バージョンの進化に伴なう警告数の変化

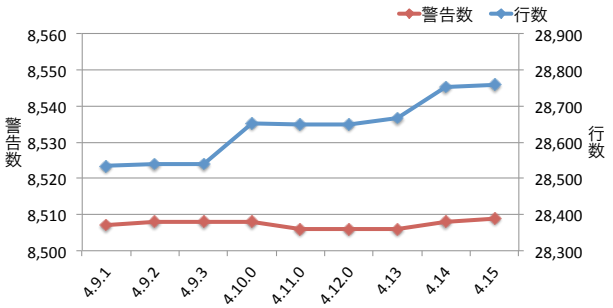
各ソフトウェアについてバージョンごとのソースコードの行数と警告数の変化を図 5 に示した．警告数の変化は行数の変化に比べて小さいため，グラフの警告数は行数の 10 倍のスケールにしている．行数はコメントや空行をカウントしない論理行数の方が適切であるが，ここで示しているデータではテキストの行数である．



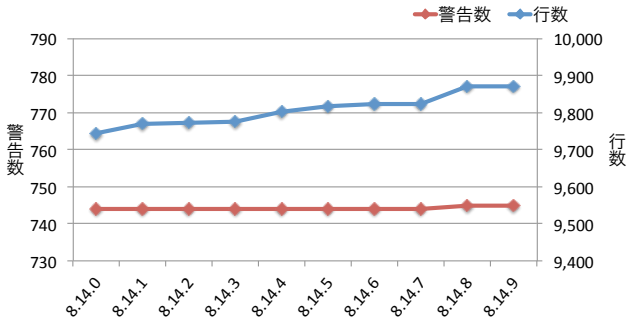
(a) OpenSSL



(b) Dovecot



(c) Courier-IMAP



(d) Sendmail

図 5: SLOC と警告数の変化

Splint による警告は，他の静的検査ツールと同様に false-positive が多く存在しているため，その多くは問題のない箇所かもしれない．しかし，各バージョンにおいて，数万

の警告が提示される．その他の静的検査ツールでも同様に多数の警告が提示され，不具合と関係ないかもしれない警告箇所を逐一確認することは現実的ではない．

OpenSSL と Dovecot では，概ね行数の増減に伴なって警告数が増減している．このことから，OpenSSL と Dovecot では Splint が出力する警告が積み残されている，もしくは警告箇所が適切に修正された数よりも多くの警告が新たに追加されていると判断できる．Courier-IMAP では警告数の最大が 8,509，最小が 8,507，Sendmail では最大が 745，最小が 744 であり，Courier-IMAP と Sendmail では警告数に大きな変化はない．このことから，新たに追加されたコードには Splint による警告がほとんどない，もしくは警告箇所が適切に修正された数と同数の警告が新たに追加されていると判断できる．これらの変化が実際にどのようになっているかは，ソースコードの変化を追跡して調査しなければわからない．

5.2 提案手法による警告箇所の削減

本調査では，前バージョンにおいて検出された警告箇所全てを確認済みとした場合に，対象バージョンにおいて新たに検出された警告箇所がいくつあるかを計測した．また，確認済みとして記録したデータのうち，警告内容には警告メッセージ中の最初の一単語のみを利用した．Splint の警告メッセージにはソースコード中のプログラム断片が埋め込まれるため，同じ警告かどうかの判定を容易にするためである．

調査手順

それぞれのソフトウェアの各バージョンに対して下記手順に従って確認済みの警告箇所をバージョンごとに記録する．

(1) git repository にソースコードを追加 (git add)
(2) git repository に登録 (git commit)
(3) splint を実行し，その結果からバージョンを識別するためのハッシュ値・ファイル名・行番号，警告メッセージの最初の 1 単語，最後に修正されたバージョンのハッシュ値・ファイル名・行番号を確認済みの警告として登録
(4) ファイルを削除
次バージョンの追加時に前バージョンとの差分を適用して更新しても良いが，処理を単純化するために一度全ファイルを削除し，新たにソースコードをコピーして登録する

次に，最初のバージョンを除く各バージョンに対し，下記条件のいずれかを満たす警告箇所を出力する．

(1) 対象バージョンで変更された行
(2) 変更されず，最後に修正されたバージョンの該当行と異なる警告

この手順によって得られた警告箇所は，前バージョンまでのいずれにおいても確認していない警告箇所であり，新たに検出された警告箇所である．

調査結果

表 1 にそれぞれのソフトウェアの各バージョンにおける警告数，総行数，新たな警告数，修正行数を示した．行数はコメントや空行も含めた行数である．修正行数は追加された行および変更された行の数である．

Dovecot では新たな警告箇所の数が 100 を超えており，それら全てを確認することは労力がかかると考えられるが，それ以外の多くは 100 以内に抑えられている．除外された警告箇所はソースコードを変更しておらず，かつ違反内容に変更がない箇所であるため，過去のバージョンで一度確認済みの問題のない警告 (静的検査ツールの誤検出) である．したがって，提案手法により提示される警告箇所は，開発者が未確認の箇所のみであり，開発者の確認コストを大幅に削減することができる．

考察

本調査では，リリースバージョン間での警告数について調査したが，実際にはコミット時などもっと高い頻度で検査することが多いと考えられる．高頻度で検査することで一回の検査時に確認すべき警告数はより少なくなる．また，本実験では修正行当たりの警告数は大きくて 0.35 程度であり，多くの場合 0.1 前後あるいは 0.1 未満である．そのため，こまめに検査することで確認すべき警告箇所が多過ぎて確認することを怠ることがなくなると考えられる．

Courier-IMAP と Sendmail では，警告数と行数の変化だけに着目すると，その変化は数個の違いであったが，ソースコードの変更を追跡することによって，変化が明確になった．ただし，Sendmail に対して新たに検出された警告箇所全てを確認したところ，その全ての変更は静的検査ツールが生成した警告とは関係ない箇所の変更であり，警告内容は同じものであった．具体例を図 6 に示す．この例では，137 行目の条件式中の ‘== NULL’ が ‘< 0’ に修正されているが，警告は関数 sm_io_fgets の第 4 引数の型の不一致についてである．このように警告内容が同じ場合には警告対象の行に変更があっても警告を修正するための変更ではないため，警告から除外すべきであるが，提案手法では新たな警告として検出してしまう．

その他のソフトウェアも同様に新たな警告箇所として提示された箇所でも同じ内容の警告箇所が多数存在すると考えられる．これらは警告箇所の前後の行の対応関係を調査することによって検出することが可能であり，これらを除外することが可能である．

本調査では，同一の種類の警告を判定処理を簡便化するために，警告メッセージの最初の一単語を用いているが，

表 1: 提案手法により検出される警告数の調査結果

(a) OpenSSL

バージョン	警告数	行数	新たな警告数	修正行数
0.9.8m	38,793	393,459	-	-
0.9.8n	38,873	393,511	6	85
0.9.8o	38,966	393,754	130	444
0.9.8p	38,939	393,693	69	198
0.9.8q	38,929	393,735	0	53
0.9.8r	38,936	393,763	9	66
0.9.8s	38,964	394,196	65	530
0.9.8t	38,964	394,197	1	20
0.9.8u	38,995	394,416	29	330
0.9.8v	39,003	394,451	9	66
0.9.8w	39,003	394,461	0	15
0.9.8x	39,003	394,461	0	7
0.9.8y	39,115	396,404	186	1291
0.9.8za	39,140	395,915	95	639

(b) Dovecot

バージョン	警告数	行数	新たな警告数	修正行数
2.2.0	39,751	344,866	-	-
2.2.1	39,772	344,945	19	101
2.2.2	39,860	346,031	110	2,956
2.2.3	40,260	348,026	499	2,865
2.2.4	40,269	348,474	22	537
2.2.5	40,334	350,053	358	2,771
2.2.6	41,079	353,540	862	4,675
2.2.7	41,474	356,298	506	3,453
2.2.8	41,591	357,008	210	1,164
2.2.9	41,718	358,657	232	2,375
2.2.10	41,814	359,955	130	1,810
2.2.11	42,016	361,419	373	3,467
2.2.12	42,016	361,419	0	9
2.2.13	42,332	364,513	567	4,596

(c) Courier-IMAP

バージョン	警告数	行数	新たな警告数	修正行数
4.9.1	8,507	28,535	0	8
4.9.2	8,508	28,541	2	24
4.9.3	8,508	28,541	0	5
4.10.0	8,508	28,651	23	937
4.11.0	8,506	28,650	1	51
4.12.0	8,506	28,650	0	72
4.13	8,506	28,666	0	390
4.14	8,508	28,753	14	341
4.15	8,509	28,760	1	125

(d) Sendmail

バージョン	警告数	行数	新たな警告数	修正行数
8.14.0	744	9,744	5	11,575
8.14.1	744	9,768	2	490
8.14.2	744	9,771	9	1,090
8.14.3	744	9,776	2	468
8.14.4	744	9,802	5	2,008
8.14.5	744	9,816	3	918
8.14.6	744	9,823	2	616
8.14.7	744	9,824	11	903
8.14.8	745	9,872	31	1,866
8.14.9	745	9,872	31	438

Splint では最初の一単語が同じ複数の異なる警告の種類が存在する．そのため，本調査で誤って同一の種類と判定されている可能性がある．同一の種類の警告を判定する方法については今後検討する必要がある．

提案手法では，版間で追跡された行で同一の警告内容である場合，その警告は確認済みとして判定している．そのため，同一行に同じ種類の警告が複数存在する場合，誤って確認済みと判定される．しかし，同一の種類の警告で，過去のバージョンで問題ないと判定された警告であれば，多くの場合，誤って確認済みと判定されても問題ないと考えられる．

6. おわりに

本稿では，開発者が静的検査ツールの結果を確認する際のコストを削減するために，過去のバージョンにおいて問題ないと確認した警告箇所を，現バージョンの結果から除外するための手法を提案した．4 つのオープンソースソフトウェアを対象に実験したところ，前バージョンの警告箇所を全て確認したと仮定すると，確認すべき警告箇所を大幅に削減することができた．

今後は提案手法を eclipse などの統合開発環境や Jenkins などの継続的インテグレーションツールで利用できるようにツールとして実現する．さらに，実装したツールを実際

```
rmail.c:137:11: Function sm_io_fgets
    expects arg 4 to be int gets size_t: sizeof((lbuf))
```

(a) 警告メッセージ

```
@@ -134,7 +134,7 @@ main(argc, argv)
   {
      /* Get and nul-terminate the line. */
      if (sm_io_fgets(smioin, SM_TIME_DEFAULT, lbuf,
-                     sizeof(lbuf)) == NULL)
+                     sizeof(lbuf)) < 0)
        err(EX_DATAERR, "no data");
        if ((p = strchr(lbuf, '\n')) == NULL)
          err(EX_DATAERR, "line too long");
```

(b) 変更内容

図 6: 同じ警告が新たな警告として提示される例

の開発に適用し，提案手法の評価を行う．また，このツールにより開発者が問題ないと判断した警告について分析し，false-positive を削減するための手法について検討する．

また，提案手法では修正の必要がない箇所のみを記録するが，修正の必要がある箇所も記録し，それらを蓄積することによって，過去に修正の必要がないと判断した箇所について，問題がある可能性がある箇所として再度提示することが可能となる．これにより，開発者が誤って修正の必要がないと判断した箇所を見直すことが可能となる．

謝辞　本研究の一部は科研費 基盤研究 (B) 24300006，若手研究 (B) 24700036 の助成を受けた．

参考文献

[1] Kernighan, B. W. and Ritchie, D. M.: *C programming language*, Prentice Hall, second edition (1988).
[2] *GNU Coding Standard*. http://www.gnu.org/prep/standards/.
[3] Association, M. I. S. R.: *Guidelines for the Use of the C Language in Critical Systems*, Motor Industry Research Association (2013).
[4] *CERT Coding Standards*. https://www.securecoding.cert.org/.
[5] 静的解析ツール QAC. http://www.programmingresearch.com/products/qac/.
[6] 大須賀俊憲，小林隆志，渥美紀寿，間瀬順一，山本晋一郎，鈴村延保，阿草清滋：CX-Chekcer: 柔軟にカスタマイズ可能な C 言語プログラムのコーディングチェッカ，情報処理学会論文誌， Vol. 53, No. 2, pp. 590–600 (2012).
[7] *Splint - Secure Programming Lint*. http://www.splint.org/.
[8] *Cppcheck A tool for static C/C++ code analysis*. http://cppcheck.sourceforge.net/.
[9] *Clang Static Analyzer*. http://clang-analyzer.llvm.org/.
[10] *PMD*. http://pmd.sourceforge.net/.
[11] *Checkstyle*. http://checkstyle.sourceforge.net/.
[12] Bland, M.: Finding More Than One Worm in the Apple, *Communications of the ACM*, Vol. 57, No. 7, pp. 58–64 (2014).
[13] Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R.: Why Don't Software Developers Use Static Analysis Tools to Find Bugs?, *Proceedings of the 2013 International Conference on Software Engineering*, pp. 672–681 (2013).
[14] Thung, F., Lucia, Lo, D., Jiang, L., Rahman, F. and Devanbu, P. T.: To What Extent Could We Detect Field Defects? An Empirical Study of False Negatives in Static Bug Finding Tools, *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 50–59 (2012).
[15] Nanda, M. G. and Sinha, S.: Accurate Interprocedural Null-Dereference Analysis for Java, *Proceedings of the 31st International Conference on Software Engineering*, pp. 133–143 (2009).
[16] Shen, H., Fang, J. and Zhao, J.: EFindBugs: Effective Error Ranking for FindBugs, *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 299–308 (2011).
[17] Heckman, S. and Williams, L.: On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques, *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 41–50 (2008).
[18] *RATS - Rough Auditing Tool for Security*. https://code.google.com/p/rough-auditing-tool-for-security/.
[19] *ITS4: Software Security Tool*. http://www.cigital.com/its4/.
[20] *Flawfinder*. http://www.dwheeler.com/flawfinder/.
[21] Chatzieleftheriou, G. and Katsaros, P.: Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities, *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pp. 96–103 (2011).
[22] Zitser, M., Lippmann, R. and Leek, T.: Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code, *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pp. 97–106 (2004).
[23] Muske, T., Baid, A. and Sanas, T.: Review efforts reduction by partitioning of static analysis warnings, *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pp. 106–115 (2013).
[24] Tripp, O., Guarnieri, S., Pistoia, M. and Aravkin, A.: ALETHEIA: Improving the Usability of Static Security Analysis, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 762–774 (2014).
[25] Arai, S., Sakamoto, K., Washizaki, H. and Fukazawa, Y.: A Gamified Tool for Motivating Developers to Remove Warnings of Bug Pattern Tools, *Proceedings of the 2014 6th International Workshop on Empirical Software Engineering in Practice*, pp. 37–42 (2014).