Software Engineering Institute

JPCERT CC®

ソースコード解析ツールを活用した CERT セキュアコーディングスタンダードの有効性評価

Stephen Dewhurst
Chad Dougherty
Yurie Ito（伊藤 友里恵）
David Keaton
Dan Saks
Robert C. Seacord
David Svoboda
Chris Taschner
Kazuya Togashi（富樫 一哉）

2008 年 6 月

Carnegie Mellon

JPCERT CC®

Notifications and Disclaimers.

The following notices and disclaimers must be contained on the unofficial SEI-sanctioned translation in both English and Japanese:

- *This translation of Carnegie Mellon University copyrighted material is not an offical SEI-sanctioned translation.*
- *This non-SEI-sanctioned translation of "Evaluation of CERT Secure Coding Rules through Integration with Source Code Analysis Tools," CMU/SEI-2008-TR-014, ESC-TR-2008-014, Copyright 2008 by Carnegie Mellon University was prepared by JPCERT/CC with special permission from the Software Engineering Institute.*
- *Neither Carnegie Mellon University nor the Software Engineering Institute directly or indirectly endorse this non-SEI-sanctioned translation. Accuracy and interpretation of this translation are the responsibility of JPCERT/CC. The SEI has not participated in this translation.*
- *CERT is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.*
- *Copyright 2008 Carnegie Mellon University.*

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

JPCERT CC®

目次

図の目次

表の目次

謝辞

本調査プロジェクトに有志として参加し C/C++ のプロジェクト評価において多大な貢献を頂いた株式会社 SRA のスタッフに感謝する。ソースコード解析ツールの提供および技術的サポートで協力頂いた米国 Fortify Software およびフォーティファイ・ソフトウェア株式会社の Brian Chess 氏、Jacob West 氏、Geoff Morrison 氏、Erik Klein 氏、John Forsythe 氏、Kannan Goundan 氏、Ning Wang 氏、遠藤玄声氏に感謝する。ローレンスリバーモア国立研究所の Daniel Quinlan 氏には Compass/ROSE を使用した調査実施にご協力頂いた。この場を借りて感謝する。本調査プロジェクトの実現に欠かせぬ貢献をしてくれた JPCERT/CC の椎木孝斉氏および久保正樹氏、SEI の Pamela Curtis 氏、Bob Rosenstein 氏、Jason Rafail 氏、Jeff Carpenter 氏に感謝する。

エグゼクティブサマリ

本報告書では、セキュアコーディング作法の有効性を評価する調査の結果を示す。セキュアコーディングスタンダード（CERT C Programming Language Secure Coding Standard [CERT 07a] および CERT C++ Programming Language Secure Coding Standard [CERT 07b]）を実装したソースコード解析ツールの利用もその作法に含まれる。

本調査プロジェクトは CERT Secure Coding Initiative と JPCERT/CC の共同の取り組みである。

CERT Secure Coding Initiative は、ソフトウェア開発に携わる個人・組織と連携して、コーディングエラーに起因する脆弱性をソフトウェアが配布される前に排除するべく設立された。脆弱性の数を世界各国に存在する脆弱性分析チームが対応できる水準にまで減らし、ソフトウェアが配布される前に脆弱性を排除することで修正コストを削減することをその取り組みの主な目的としている。

JPCERT コーディネーションセンター(JPCERT/CC)は、情報システムの円滑な運用の脅威となるコンピュータ セキュリティ インシデントに対応する組織（CSIRT：Computer Security Incident Response Team）であり、①コンピュータの不正利用などによるインシデントへの対応、②ワームの感染活動の観測をはじめとするインターネット定点観測システムの運用、③ソフトウェアの脆弱性に関する調整、④コンピュータセキュリティインシデントを未然に防ぐための早期警戒活動など、情報セキュリティ対策活動のコーディネーションを行っている。また、国内における技術情報の配信やイベントを通じた啓発活動、およびアジア太平洋地域における CSIRT 間の情報交換網の構築や組織間の連携強化を主導している。

本調査プロジェクトの目的は、商用ソフトウェアプロジェクトの品質とセキュリティの向上における、CERT セキュアコーディングスタンダードとソースコード解析ツールの有効性を評価することである。ソースコード解析ツールには、拡張性と有効性の観点から、Fortify Software の Fortify Source Code Analyzer（SCA）とローレンスリバーモア国立研究所の Compass/ROSE の 2 つのツールが選ばれた。次に CERT C/C++ セキュアコーディングスタンダードへの違反を検出するための拡張チェッカーモジュールが開発された。ツールはその後、日本の大手ソフトウェア開発会社である株式会社 SRA に渡された。SRA は拡張チェッカーモジュールを搭載したバージョンの Fortify SCA および Compass/ROSE を 2 つのソフトウェア開発プロジェクトで評価した。その 2 つのプロジェクトとは、C++言語で書かれた 料金徴収システム関連の GUI アプリケーションと、C 言語で書かれた 映像サービスの通信プロトコルの開発プロジェクトである。

ソースコード解析ツールの拡張は成功し、2 つの評価対象開発プロジェクトで多くのソフトウェアの欠陥を検出し、CERT セキュアコーディングスタンダードとソースコード解析ツールの双方においてソフトウェアの品質向上における有効性が証明された。また、CERT セキュアコーディングスタンダードとソースコード解析ツールの両方をさらに改良する方法も特定することができた。

アブストラクト

本報告書は、商用ソフトウェアプロジェクトの品質とセキュリティの向上における CERT セキュアコーディングスタンダードとソースコード解析ツールの有効性を評価するために、CERT Secure Coding Initiative および JPCERT/CC が実施した調査についてまとめたものである。既存のソースコード解析ツールが CERT セキュアコーディングスタンダードへの違反を検出できるかどうかを評価するとともに、ツールの拡張と改良の可能性も調査する。最後に、日本のソフトウェア開発会社の実際の製品開発プロジェクト事例を用いて、選択したツールを使ったソースコードの品質向上について述べる。

1 調査プロジェクトの概要

本報告書は、CERT C Programming Language Secure Coding Standard [CERT 07a] および CERT C++ Programming Language Secure Coding Standard [CERT 07b] に記載されているセキュアコーディングスタンダードにおいて定義されたルールおよび推奨事項の有効性を評価するための調査結果を示したものである。

1.1 CERT セキュアコーディングスタンダードの概要

ネットワーク化されたソフトウェアシステムへの依存度が高まるにつれ、そのようなシステムを狙った攻撃の数は増えてきている。政府、企業、教育機関、個人を狙ったこれらの攻撃は、機密データの消失や漏えい、システムの損傷、生産性の低下、および金銭的な損失につながっている [Seacord 05]。

ソフトウェアの脆弱性に関する報告は驚くべき速さで増え続けており [CERT 07c]、そのうちの大部分はテクニカルアラート（注意喚起）の発行につながっている [US-CERT 08]。この増大する脅威に対処するためには、ソフトウェアの開発や継続的なメンテナンスの過程で作りこまれるソフトウェアの脆弱性を大幅に削減する必要がある。

安全なソフトウェア開発に不可欠な要素は、適切に明文化された、適用可能なコーディングスタンダードである。コーディングスタンダードがあれば、プログラマは個人の得意な方法や好みではなく、プロジェクトと組織の要件によって決められた統一的な一連のルールとガイドラインに従うことができる。一度標準が策定されれば、それを（手動または自動のプロセスにより）ソースコードの評価基準として使用し、標準への適合を確認することができる。

CERT が提案しているセキュアコーディングスタンダードは、標準化団体が定める明文化された標準言語に基づく。たとえば、CERT セキュアコーディングスタンダードは次の言語を対象としている。

- プログラミング言語 C（ISO/IEC 9899:1999）[ISO/IEC 9899-1999]
- プログラミング言語 C++（ISO/IEC 14882:2003）[ISO/IEC 14882-2003]

ISO/IEC TR 24731 C 言語ライブラリ拡張 [ISO/IEC TR 24731-1-2007] のような技術正誤表や言語拡張へ、 CERT セキュアコーディングスタンダードを適用することも検討されている。

適用範囲としては、特定の指針を広範なユーザ層に提供することを念頭に置いている。ISO/IEC が策定するプログラミング言語の標準などは、主にコンパイラ実装者を対象に書かれているが、CERT セキュアコーディングスタンダードは、それら標準で定められた言語でプログラムを書く開発者を直接の対象として書かれたルールと推奨事項を示す補助的な文書である。

C 言語、C++言語用の CERT セキュアコーディングスタンダードの目的は、それぞれのプログラミング言語で開発されたソフトウェアシステムのセキュリティを確保するために必要な（ただし十分ではない）一連のルールを定めることである。

CERT セキュアコーディングスタンダードは、**ルール**と**推奨事項**から構成されている。次の３つの条件をすべて満たすコーディング作法は、ルールとして定義されている。

1. コーディング作法に違反することが、攻撃可能な脆弱性を生み出すセキュリティ上の欠陥ににつながる可能性がある。
2. プログラムが正しく動作するには、コーディング作法に違反せねばならない一連の有限の条件が存在する（あるいは存在しない）。
3. コーディング作法へ適合しているかどうかを、自動解析、形式的方法、手作業によるソースコード検査などを通じて確認することができる。

ある特定のルールに例外条件が存在する場合を除き、標準に適合するためにはそのルールに従わねばならない。ルールに従わずに例外条件を適用する場合、その例外はあらかじめ定義された例外条件と一致する必要があり、この例外の適用はソースコード内に記されている必要がある。

推奨事項は、ガイドラインまたは提案である。コーディング作法は、次の２つの条件をすべて満たすとき推奨事項として定義されている。

1. コーディング作法を適用することで、システムのセキュリティが高まる可能性がある。
2. コーディング作法がルールと見なされるための要件を１つ以上満たすことができない。

推奨事項への適合は CERT セキュアコーディングスタンダードに適合するための必要条件ではない。しかし、1 つまたは複数の検証可能なガイドラインに適合していると言える。特定の開発作業で適用する一連の推奨事項は、最終的なソフトウェア製品のセキュリティ要件によって異なる。高いセキュリティが要求されるソフトウェア開発プロジェクトでは、より多くのリソースをセキュリティ対策に投入でき、結果的により多くの推奨事項を適用することが考えられる。

1.2 本調査プロジェクトのスケジュールの概要

図 1 は、本報告書で説明する調査の概要を示している。プロジェクトは 2007 年 8 月から 2008 年 3 月までの間に実施され、4 つの主要なタスクで構成された。

図1：本調査プロジェクトのスケジュール

本調査プロジェクトの第 1 段階では、CERT Secure Coding Initiative（SCI）が、調査で使用するソースコード解析ツールを評価した。第 2 段階では、CERT SCI が、第 1 段階で選ばれたソースコード解析ツール用の拡張チェッカーモジュールを開発した。この最初の 2 つの段階については、本報告書のセクション 2 で詳しく述べる。開発された拡張チェッカーモジュールは、ソフトウェア開発プロジェクトにおける CERT セキュアコーディングスタンダードへの違反を検出・評価するために JPCERT/CC と SRA に提供された。最後に、結果の分析と最終報告の準備に 1 ヶ月が予定された。

2　ソースコード解析ツールの評価とチェッカー開発

本調査プロジェクトの第 1 段階では、既存の商用および非商用のソースコード解析ツールの本プロジェクトへの適合性を評価した。用いた評価プロセスは、Software Engineering Institute で過去に採用されたファーストフィットのアプローチである [Wallnau 01]。評価基準には以下の項目が含まれるが、これに限られるわけではない。

- **有効性：**業界で一般的な手法に影響を与えるには、既存のベストプラクティスをすでに反映しているソースコード解析ツールから開始する必要がある。このツールをさらに強化することで、手法の水準を高められると考えられた。
- **拡張性：**ソースコード解析ツールでサポートされる既存のルールセットを拡張し、CERT セキュアコーディングスタンダードのサポートを盛り込む必要があるため、拡張性は重要な評価基準である。
- **適合性：**このソースコード解析ツールは特定のソフトウェア開発者によって特定の状況で使われるため、開発者とその選択したプロジェクトのニーズに合う必要がある。いくつかある特徴の中でも、開発者が使用するプラットフォームとコンパイラにおける可用性と、ソフトウェア開発者の好みが、適合性として考慮される。

本調査プロジェクトでは次のソースコード解析ツールを評価した。

- Fortify SCA バージョン 4.5（バージョン 4.5 で拡張チェッカーが作成され、その後バージョン 5.0 で実際に分析が行われた）
- ローレンスリバーモア国立研究所（LLNL）の Compass/ROSE

2.1　FORTIFY SCA の評価とチェッカー開発

Fortify Source Code Analyzer（SCA）は、各種のプログラミング言語（たとえば Java、C、C++）を中間形式に変換する。この中間形式はさらに、危険なコーディング構造を識別・検出するための一連のアルゴリズムを使って処理される [Chess 02]。Fortify SCA は 5 つの分析エンジンで構成されている。この 5 つの分析エンジンとは、データフローアナライザ（data flow analyzer）、制御フローアナライザ（control flow analyzer）、セマンティックアナライザ（semantic analyzer）、構造アナライザ（structural analyzer）、コンフィギュレーションアナライザ（configuration analyzer）であり、これらを組み合わせてソースコードの脆弱性を特定する。これらの分析エンジンはそれぞれ異なる方法で脆弱性を検出する。

データフローアナライザは、ユーザによって提供されるデータを、その入力から関数を通じてプログラムに伝播するまで追跡しようと試みる。このアナライザの目的は、ユーザが提供するデータの危険な使用が原因で発生する脆弱性を特定することである。グローバルなプロシージャ間の汚染伝播を追跡して関数呼び出しや演算内の脆弱性を検出する。

制御フローアナライザは、単一の関数またはメソッドにおける一連の操作にみられる脆弱性を探す。アナライザは、与えられたソースコードにステートマシンを適用することで、危険な動作を明らかにする。プログラムの制御フローパスを解析し、演算が一定の順序で実行されているかどうかを確認する。アナライザはこれらの方法を使い、一連の危険な関数呼び出しを特定できるだけでなく、結果として関数が呼び出されない原因となる一連の操作における脆弱性を検出できる。

セマンティックアナライザは、危険な関数呼び出しを見つけるためにシグネチャを使用する。アナライザは、脆弱性を検出するために、プロシージャ間レベルでの関数やアプリケーショ

ンプログラミングインタフェース（API）の使用を分析する。この分析エンジンには、一般的な問題の検出に加え、バッファオーバーフロー、書式指定文字列、実行パス脆弱性の検出など特殊な手法も含まれている。これらの技法を用いて、セマンティックアナライザは、潜在的に危険な関数呼び出しを検出できる。

構造アナライザは、ソースコードの構成上の脆弱性を探す。この分析エンジンは、Fortify SCA が提供する分析エンジンの中でもっとも柔軟性があり、かつ強力な分析を行える可能性がある。この分析エンジンによって、命令行とその命令行を含むソースコードブロックの関係に、潜在的な脆弱性がないか注意深く調べることができる。構造アナライザは、変数や関数の宣言と使用の両方を含む範囲が分析対象となるがゆえ、検出が困難なことが多いセキュアプログラミング作法や技法の違反を構造アナライザは検出できる。

コンフィギュレーションアナライザは、テキストのプロパティファイルや XML 設定ファイルの脆弱性を検出する。このアナライザは、危険なキー値の組み合わせや安全でない XML 要素および属性定義を特定しようとする。

ソースコードは Fortify SCA の Fortify プログラムパックに含まれる `sourceanalyzer` プログラムを使用してスキャンされる。一連のコマンドの例を次に示す。

```
sourceanalyzer -b sci-a gcc -g3 -o a a.c

sourceanalyzer -b sci-a -scan -rules sci-rules.xml –rules \

  sci-structure-rules.xml -level broad -logfile filename.log
```

Fortify SCA のソースコード解析プロセスは、変換、解析、検証の 3 つの段階で構成されている。

解析プロセスの変換段階では、Fortify SCA は一連のコマンドを通じてソースコードを収集する。次に、ソースコードはユーザ指定のビルド ID が関連付けられた中間形式に変換される。ビルド ID は、Fortify SCA がチェックするプロジェクトで一意となる必要がある。

変換段階の次は、解析段階である。この段階では、変換段階で特定されたソースファイルがチェックされ、Fortify SCA が解析結果ファイルを生成する。

変換、解析段階のあとに続くのが、検証段階である。ユーザは、Fortify SCA が報告した深刻なエラーを解析ファイルで調べる。エラーは次の形式をとる。

```
[<ルール ID 番号> : <深刻度> : <ルールタイトル> : <分析エンジン>]

      <ソースファイル名>(<行番号>) : <詳細>
```

Fortify SCA の各分析エンジンは拡張可能である。各分析エンジンにはカスタムルールを記述することができる。カスタムルールは XML で記述して、それ単体で、あるいは Fortify SCA にあらかじめ組み込まれたルールセットと一緒に使うことができる。

全 180 個の CERT C セキュアコーディングスタンダードの内 で 113 個のルールと推奨事項、そして 、全 99 個 の CERT C++ セキュアコーディングスタンダードの 内で 63 個のルールと推奨事項について、拡張チェッカーを加える前の Fortify SCA でどの程度の CERT C/C++ セキュアコーディングスタンダードへの違反が検出可能であるかの調査が行われた。これらのルールと推奨事項への違反検出を確認するためのサンプルソースコードを Fortify SCA を用いて解析を行った。Fortify SCA は分析エンジンの追加的な拡張なしで、これら CERT C セ

キュアコーディングスタンダードのルールと推奨事項の内 16 個と CERT C++ セキュアコーディングスタンダードの ルール・推奨事項の内 6 個に対する違反を検出した。

Fortify SCA が他の CERT C/C++ セキュアコーディングスタンダードの違反を検出できるように、カスタムルール（拡張チェッカー）を作成して分析エンジンを拡張した。このような拡張チェッカーは、制御フローアナライザ、セマンティックアナライザ、構造アナライザを拡張する。制御フローアナライザ拡張のために 18 個、セマンティックアナライザ拡張のために 9 個、そして構造アナライザ拡張のために 31 個のルールと推奨事項への違反を検出するための拡張チェッカーがそれぞれ作成された。[1]

これらの拡張チェッカーが Fortify SCA に追加されると、Fortify SCA は 27 個の CERT C セキュアコーディングスタンダードの ルール・推奨事項と、 17 個の CERT C++ セキュアコーディングスタンダードのルール・推奨事項のすべての側面を捉えることができた。Fortify SCA は、他の 14 個の CERT C セキュアコーディング スタンダードのルール・推奨事項と 、5 個の CERT C++ セキュアコーディングスタンダードのルール・推奨事項については、一部の側面しか検出できなかった。全体として、Fortify SCA は、調査した CERT C/C++ セキュアコーディングスタンダードのルール・推奨事項の 176 個の内 少なくとも 85 個を部分的に捉えることができた。

	C	C++	合計
CERT セキュアコーディングスタンダードのルール・推奨事項の総数	180	99	279
Fortify SCA にて違反検出が可能かを確認したルール・推奨事項の数	113	63	176
拡張チェッカーを追加前の Fortify SCA で違反検出が可能であったルールと推奨事項の数	16	6	22
拡張チェッカーを Fortify SCA 用に新たに追加することで、違反検出が可能となったルールと推奨事項の数	27	17	44
拡張チェッカーを Fortify SCA 用に新たに追加することで、部分的に違反検出が可能となったルールと推奨事項の数	14	5	19
少なくとも部分的に違反検出することが可能となったルールと推奨事項の総数	57	28	85

表 1：Fortify SCA を使用した場合の CERT セキュアコーディングスタンダードに対する違反検出可能ルール・推奨事項の数

拡張チェッカーを追加前の Fortify SCA では検出できなかった CERT C/C++ セキュアコーディングスタンダードのルール・推奨事項への違反を検出するために Fortify SCA 用の拡張チェッカーを開発中に実装上いくつかの制約が見られた。これらの制約は、拡張チェッカーの違反検出精度や、拡張チェッカーとして実装可能なルール・推奨事項の数に影響した。まず、各 CERT C/C++ セキュアコーディングスタンダードのルール・推奨事項に対して適合していないソースコードすべてを検出する拡張チェッカーを作成する試みが行われたが、ルール・推奨事項に記述されたすべての側面を検出できない場合には、適合していないソースコードの一部あるいは大部分を検出する方針で拡張チェッカーは作成された。

Fortify SCA の解析のタイミング、そして解析されたソースコードの帰結性が、本調査プロジェクトにおいて FortifySCA 用に記述した拡張チェッカー多くに制限をもたらした。Fortify

[1] C および C++ スタンダードの両方に登場するルールがいくつかある。重複は、分析エンジンによって、この総数から削除されている。

SCA はコンパイル前にソースコードを検査しない。逆に、Fortify SCA は、前述の変換段階の結果を検査する。解析は変換後に行われるので、いずれの分析エンジンも元のソースコードのすべての構文パターンを照合することはできない。

変換後に解析を行うという性質ゆえ、Fortify SCA は CERT セキュアコーディングスタンダードの多くの側面を検出することができない。Fortify SCA は「プリプロセッサ」と「シグナル」のカテゴリのルールはまったく検出できなかった。Fortify SCA がソースコードを解析する前に、すべての `for` ループは `while` ループに変換される。`sizeof` 演算子、`const` 修飾子、`enum` 指定子はすべて変換段階において削除される。複数の識別子定義は、変換段階で自動的に解決される。

Fortify SCA のもう 1 つの制約は、関数の境界によって解析が限定されることである。Fortify SCA は、境界を越えて汚染、定数、プログラムの状態を追跡するが、グローバルな型状態解析は行わない。これは、複数の関数の相互作用の仕方が原因で発生する問題への対処を目的として定義されたすべてのセキュアコーディングスタンダードのルール・推奨事項に影響する。これには、存続期間外のオブジェクトの参照やファイルストリームの再オープンに関連する問題などが含まれる。

Fortify SCA の配列処理の制約も障害となる。Fortify SCA の変換段階で、配列はポインタに変換される。この変換により、配列に固有のルールはすべてのポインタに反応し、結果として非常に多くのフォールスポジティブ（誤検出）を引き起こす。

また、条件式も問題をもたらした。Fortify SCA は構造分析エンジン内の条件式を無視する。これは、条件式を検出するために作成したルールを、ほかのいずれかの分析エンジンで記述しなければならないということである。その結果、本来検出されるはずの違反件数が限定された。

制御フロー分析エンジンでの変数追跡も問題であった。Fortify SCA は変数ではなくデータを追跡する。このため、変数に 0 が割り当てられた場合、ステートマシンはその変数の追跡を停止する。これは、変数を追跡しようとするルール内にフォールスネガティブ（検出漏れ）を引き起こす。

制御フローアナライザ、セマンティックアナライザ、構造アナライザ拡張のための Fortify SCA 向け拡張チェッカー作成作業は、Fortify の技術スタッフとの継続的な協力のもと行われた。

付録 A と付録 B において、特定の CERT セキュアコーディングスタンダードのルール・推奨事項に対するチェッカーの実装詳細について説明する。

2.2 COMPASS / ROSE の評価とチェッカー開発

ROSE はソースからソースへのソースコード変換を行うフレームワークである。Compass は ROSE の次のリリースに含まれる予定だが、現在のリリースでも動作する。

Compass は既存のチェッカーを使って任意のソースコードを評価するために設計されたオープンソースのツールである。ユーザ独自のドメイン固有なルールセットを使用できるように、Compass を設定することも容易である。また、Compass の拡張は容易であり、次の手順を踏むことでチェッカーを追加できる。

1. チェッカーのソースコードテンプレートを生成するスクリプトにチェッカーの名前を追加する。
2. 10 数行程度のソースコードを生成されたチェッカーコードテンプレートに記述し、コードパターンを定義する（定義は、C++言語で記述され ROSE からの高レベルな AST IR ノードを使用する）。記述するソースコードの量は実装対象となるルールによって異なる。
3. チェッカーコードテンプレートによって生成された Latex ページを使用して、チェッカーを文書化する。
4. 問題（対象ルールへの違反）を示すテストコードを用意する（このコードは文書化およびテストで使用される）。
5. ディレクトリ全体を tar 形式にし、スクリプトを実行してチェッカーをサブミットする。（このスクリプトが、チェッカーの tar ボールをすべての他のチェッカーが格納されている共有ディレクトリにコピーする）。

サブミット（チェッカーの投稿）を受けつけて処理する側において、すべてのサブミットされたチェッカーは自動的に Compass に組み込まれる。

1. すべてのチェッカーの tar ボールを、既存バージョンの Compass に自動的に組み込むスクリプトを実行する。
2. プロセスを完了するには "make" を実行し、文書を作成するには "make docs" を実行する。全プロセスをテストするには "make test" を実行する。

このプロセスは多くのユーザからのチェッカーの投稿を支援する目的で設計された。Compass はテキストモード、GUI モードのどちらでも実行できる。GUI モードはロンドン大学インペリアルカレッジ（元学生が Qt ベースで、ROSE 向けの GUI ビルダを開発した）の協力で作成された。

例えば、Compass/ROSE は、次のアルゴリズムを使って SIG30-C（「シグナルハンドラ内では async-safe 関数のみを呼び出すこと」）に適合しているかどうかをチェックできる。

1. async-safe 関数の初期リストがあるとする。POSIX は一連の関数が async-safe であることを要求するが、このリストは各 OS に依存する。
2. async-safe の特性を満たしたアプリケーションが定義するすべての関数を、async-safe 関数リストに追加する。関数は次の場合に async-safe の特性を満たす。（a）async-safe 関数リスト内の関数のみを呼び出している、（b）`sig_atomic_t` 型の volatile 静的変数に値を割り当てる以外は外部変数を参照したり変更したりしない。このことは、async-safe 関数であるシグナルハンドラ内の関数を呼び出すプロシージャ間の事例に対応する。
3. 抽象構文ツリー（AST）を走査して、シグナル関数 `signal(int, void (*f)(int))` の関数呼び出しを特定する。
4. signal(int, void (*f)(int)) の各関数呼び出しで、引数リストから 第 2 引数を取得する。これがオーバーロードされた関数ではないことを確かめるために、関数の型シグネチャが評価され、さらに／または関数の宣言個所が正しいファイルからのものであることが検証される（これはリンク時の解析ではないので、ライブラリ実装についてはテストできない）。signal() の定義はライブラリ内にあるはずなので、アプリケーション内にある signal() の定義はすべて疑わしい。
5. 登録されたシグナルハンドラに対してネスト化したクエリを実行し呼び出された関数のリストを取得する。呼び出された各関数が `async-safe` 関数リストにあることを確認する。各関数を繰り返し確認するのを避けるため、関数の最初のテスト結果を格納しておく必要がある。
6. 検出されたすべての違反を報告する。

パフォーマンス上の理由のため、Compass/ROSE のコードパターンは AST で直接指定される。制御フローや ROSE のその他のプログラム解析グラフ（呼び出しグラフ、SDG など）でパターンを指定することもできる。これは、ROSE に固有のソリューションであるため、理想的ではないが、複雑なコードパターンを指定するために必要な詳細度が得られる（主な理由は、ROSE は AST を正規化しないため、ソースレベルのすべての詳細度が提供されるためである）。

属性文法ベースの AST 走査の設計では、1 回の走査で、AST の見かけ上別の走査を約 110 倍の高パフォーマンスで実行することができる。また、最近の研究では、効率的にさらなる並列化ができる方法（マルチコア最適化）が示されている。

ROSE は使いやすく、直観的なアーキテクチャであり、簡潔なルールを素早く実装することが可能である。また ROSE は、コンパイル済みコードに関する膨大な情報を保持しており、字下げやコメントなどの項目をチェックするルールまで記述できる。

ROSE には次のような欠点があるが、深刻だとは見なされていない。

1. ドキュメントの構成は優れているが内容は不完全である。そのため、あるものが何なのかまたは何をするのかを判断するために頻繁にソースコードを読む必要がある。
2. ROSE の型システムモデルの構造は低水準で、いくぶん不明りょうである。これは、主に型クラスのオブジェクト指向の性質が原因である。（この問題への対応として、我々は ROSE の型システム上に新たな薄いレイヤーを記述することで、型システム上の実装の変更から、チェッカーの実装を分離し、取り扱いを容易にした。これをさらに拡張して一般化されたユーティリティ関数を記述し、 AST についてより高度な情報を提供することも可能である。）
3. ROSE のデータ構造は簡単には解明できない。ROSE では、AST をグラフまたは属性付きノードリストとして表示するユーティリティを使ってこの問題を軽減しているが、データ構造があまりに複雑で流動的であるためユーティリティですべてを表示できない。
4. const メンバ関数の使用に関して、いくつか小さいが気になる実装上の問題がある。
5. マクロ処理は構文解析よりも前の段階で行われるため、ROSE のマクロの使用を認識する能力はまだ多分に実験段階である。
6. バグがいくつか残っている。
7. 実装が部分的に流動的なままであるため、現在推奨されている用法が必ずしも明白ではない。

これらの欠点はあるものの、全体としてツールの品質は良い。ツールにより精通すれば、かなり素早くチェッカーを記述できるはずである。ROSE 向けのチェッカー開発技術 は努力すれば習得可能であるが、最初のハードルは高い。

なお、前述した問題（2）に対する 修正方法の 1 つは、型システム上に新たな薄いレイヤーを記述することで、型システム上の実装の変更から、チェッカーの実装を分離し、取り扱いを容易にすることである。これをさらに拡張して一般化されたユーティリティ関数を記述し、AST についてより高度な情報を提供することも可能である。

3　ソースコード分析

本調査プロジェクトにおいて、ソースコード解析ツールを使用し、ソフトウェア開発プロジェクトの分析とその評価を担当する商用ソフトウェア開発会社として、株式会社 SRA（SRA）が選ばれた。1967 年に設立された SRA は、日本でも有数の歴史と規模を誇る独立系ソフトウェア会社である。SRA は、多くの通信、データベース、インターネット/World Wide Web アプリケーションを配布、サポートしており、ソフトウェア製品市場で確固とした地位を占めている。

SRA は Fortify SCA および Compass/ROSE の拡張版を、表 2 に示す 2 つのプロジェクトで評価した。

プロジェクト	言語	OS	コンパイラ	サイズ（KLOC）
料金徴収システム関連の GUI アプリケーション	C++ （Qt ベース GUI）	Miracle Linux	GCC 3.2.3	264
映像サービスの通信プロトコル	C	Cent OS（Linux）	GCC 3.4.6	30

表 2：２つのソースコード解析ツールを用いて分析作業を実施したソフトウェア開発プロジェクトの概要

この 2 つのソフトウェア開発プロジェクトは、民間企業向けに特定のソフトウェア製品を開発することを目的としており、その製品は実運用環境で使用されることを意図している。

3.1　ソースコード解析ツールを用いたソースコードの分析作業

2 つのソフトウェア開発プロジェクトのソースコードは、それぞれ CERT C/C++ セキュアコーディングスタンダードへの違反を検出可能なチェッカーを搭載したバージョンの Fortify SCA と Compass/ROSE を使って分析された。それぞれのツールは多くの欠陥候補を見つけ、これらの欠陥候補は**ポジティブ（検出）**と分類された。各プロジェクトに精通した開発者が時間と能力の限界まで各ポジティブを分析し、それが**トゥルーポジティブ（正検出）**なのか、**フォールスポジティブ（誤検出）**なのか、または**フォールスネガティブ（検出漏れ）**なのかを判断した。2 つのツールは、ソースコードがルール・推奨事項に違反していないという肯定的判定を行わないため、トゥルーネガティブ（本当に違反が無いケース）の概念は、どちらのツールでも欠陥候補として特定されなかったソースコードに該当する。

トゥルーポジティブはツールによって正しく特定された欠陥候補（正検出）であり、フォールスポジティブは誤って特定された欠陥候補（誤検出）である。フォールスネガティブは特定のツールによって発見されなかった欠陥候補（検出漏れ）であり、追加ツールの使用、ソースコードレビュー、テストなど別の方法で見つける必要がある。本調査プロジェクトにおいて特に注目するフォールスネガティブは、２つのツールの内、一方ではトゥルーポジティブであるとされたが、他方のツールでは検出されなかった欠陥候補である。すべての場合において、分析を担当した開発者は欠陥候補がどのように検出されたかを記録した。トゥルー

ネガティブは、ルール・推奨事項に違反していないソースコードを誤ってフォールスポジティブとして特定する他のツールと比較して、これらのツールは誤検出しなかったことを示している。

本調査プロジェクトにおいて、すべてのトゥルーポジティブは**ソフトウェアの欠陥**である。また、２つのソースコード解析ツールは CERT セキュアコーディングスタンダードへの違反を検出するように拡張されていることから、トゥルーポジティブとして判断された欠陥候補は少なくとも潜在的に**セキュリティ上の欠陥**であると推測する。分析の結果発見されたセキュリティ上の欠陥が、現実に**セキュリティの脆弱性**となるかどうかは、本調査プロジェクトの範囲外である [Seacord 05]。筆者らは製品のソースコードにはアクセスできなかったため、結果の精度は SRA のスタッフの判断に依存する。

時間的制約のため、すべての CERT C/C++ セキュアコーディングスタンダードのルール・推奨事項を、拡張チェッカーとして実装することはできなった。そのため、チェッカーとして実装する CERT C/C++ セキュアコーディングスタンダードのルール、および、推奨事項は選別されなければならなかった。

CERT C セキュアコーディングスタンダードの各ルールには高、中、低の優先度が設定されている。優先度の高いルールとは、違反すると攻撃可能なコードになる可能性があり、その攻撃が深刻な結果を生む可能性があるルール（たとえば、攻撃者が任意のコードを実行できる）、または手動で修正するには困難が伴う、あるいは費用がかかるルールである。CERT C++ セキュアコーディングスタンダードのルールについては、優先度設定作業が完了しておらず、ルール・推奨事項の中には優先度が設定されていないものが多く存在する。

Fortify SCA は、CERT C セキュアコーディングスタンダードのルールへの違反を可能な範囲で、すべて検出できるように拡張された。まず、すべてのルールへの違反が 拡張チェッカーを加える前の Fortify SCA を用いて検出可能であるかを調査し、検出可能な違反すべてを特定した。その後、検出することができなかった CERT C セキュアコーディングスタンダードのルールは、それぞれ Fortify SCA 用の拡張チェッカーとして実装することが可能であるか評価された。

同じく、Fortify SCA は、CERT C++ セキュアコーディングスタンダードのルールへの違反を可能な範囲で、すべて検出できるように拡張された。まず、すべてのルールへの違反が 拡張チェッカーを加える前の Fortify SCA を用いて検出可能であるかを調査し、検出可能な違反すべてを特定した。その後、検出することができなかった CERT C++ セキュアコーディングスタンダードのルールは、それぞれ Fortify SCA 用の拡張チェッカーとして実装することが可能であるか評価された。

CERT C/C++ セキュアコーディングスタンダードのルールが評価された後に、別途推奨事項の評価が行われた。推奨事項の評価作業は、CERT C セキュアコーディングスタンダードから開始し、同セキュアコーディングスタンダードに登場する順に、すべての C と C++ 言語の推奨事項を評価する試みがなされたが、時間的制約により、すべての推奨事項に対して評価を行うことはできなかった。なお、実装された C 言語用の推奨事項の中で、C++言語用の推奨事項として同様に適用可能であるものについては、その旨の注釈が加えられた。

ROSE 用 CERT C セキュアコーディングスタンダード拡張チェッカーの開発は、優先度が高いルール・推奨事項にフォーカスして行われた。スケジュール上の制約のため、優先度が高いと判断されたルール・推奨事項の中で約半分が実装された。実装難易度が高いと考えられたものや ROSE を拡張した場合でも検出が困難なものは実装対象から除外された。

一方、CERT C++ セキュアコーディングスタンダードのルール・推奨事項については、各ルール・推奨事項の優先度設定作業中であり、多くのルール・推奨事項にはまだ優先度が設定されていない状況である。よって、優先度に基づいた CERT C++ セキュアコーディングスタンダードへの違反を検出するチェッカーの実装判断を行うことができなかった。ROSE 用にチェッカーとして実装された CERT C++ セキュアコーディングスタンダードのルール・推奨事項には次のようなものが含まれる。

- 明らかに実装することが容易なルール（たとえば **EXP03-A: & 演算子をオーバーロードしないこと**）。
- 型情報やクラス階層解析に依存するルール（たとえば **OBJ03-A: 仮想関数のオーバーロードを避けることが望ましい**）。
- 難易度は高いが実装することが可能なルール（たとえば **ERR01-A: 例外に特殊用途の型を使うこと**）。

ROSE 用に実装された CERT C/C++ セキュアコーディングスタンダードのルールと推奨事項については、付録 C、付録 D に記載されている。

4　ソースコードの分析結果とその評価

このセクションでは、Fortify SCA と ROSE 双方のソースコード解析ツールを使った 料金徴収システム 関連の GUI アプリケーションと 映像サービスの通信プロトコル開発プロジェクトの分析結果とその評価について記述する。この作業においては、２つのソースコード解析ツールに拡張実装されたチェッカーに焦点を当てており、拡張前のそれぞれのソースコード解析ツールが既に備えているチェッカーについては詳細な評価は行われていない。なお、ROSE よりも Fortify SCA 用に多くのチェッカーが実装されたが、その主な理由は、ROSE を本調査プロジェクトに加えることの適性を評価中に、チェッカーの実装作業が開始されたためである。C 言語、C++言語用合わせて、ROSE には計 27 の拡張チェッカーが実装されたのに対し、FortifySCA には計 61 個が実装された。

表 3 に本調査プロジェクトにおいて、検査されたチェッカーの総数を言語別、ツール別に示すと同時に、調査対象となった各開発プロジェクトにおいて実際に検出結果を分析したチェッカー数を示す。例えば、C++言語の開発プロジェクトである料金徴収システム関連 GUI アプリケーションにおいては、23 個の C++言語用チェッカーによる違反検査が行われ、その結果 8 つのチェッカーが違反を検出、その検出結果全て（8 つ）の有効性が分析・評価されたことを示している。

開発プロジェクト	言語	ツール	チェッカーグループ	チェッカー		
				検査された	検出された	分析された
料金徴収システム関連 GUI アプリケーション	C++	Fortify	**CERT 拡張**	**23**	**8**	**8**
			デフォルト	?[2]	21	0
		ROSE	**CERT 拡張**	**15**	**6**	**3**
			デフォルト	?[2]	14	0
映像サービスの通信プロトコル	C	Fortify	**CERT 拡張**	**38**	**20**	**7**
			デフォルト	?[2]	23	0
		ROSE	**CERT 拡張**	**12**	**3**	**3**
			デフォルト	?[2]	6	0

表 3：本調査プロジェクトにおいて分析されたチェッカーの開発プロジェクト別、ツール別の内訳

4.1　FORTIFY SCA を用いたソースコード分析の結果

料金徴収システム関連の GUI アプリケーションおよび 映像サービスの通信プロトコル開発プロジェクトにおいて Fortify SCA によって検出された CERT C/C++ セキュアコーディング

[2] ルール・推奨事項のリスト化やグループ化の方法が原因で、正確にいくつのルール・推奨事項が Fortify SCA に標準装備されているのかを判断するのが困難である。本調査プロジェクトにおいて不可欠ではなかったので、最終的にこの値を割愛した。デフォルトの ROSE のルールの数も同様の理由で割愛した。

スタンダードのルール・推奨事項への違反検出レポートの総数を表 4 に示す。違反検出数はプロジェクトおよびチェッカーグループごとにまとめて示している。「デフォルト」と示されたチェッカーグループは、Fortify SCA に標準装備されているチェッカーを示し、「CERT 拡張」と示されたチェッカーグループは、本調査プロジェクトにおいて FortifySCA を拡張するために新たに開発された CERT C/C++ セキュアコーディングスタンダードのルール・推奨事項への違反検出用チェッカーである。

開発プロジェクト	言語	チェッカーグループ	違反検出数
料金徴収システム関連の GUI アプリケーション	C++	**CERT 拡張**	**186**
		デフォルト	572
映像サービスの通信プロトコル	C	**CERT 拡張**	**408**
		デフォルト	396

表 4：Fortify SCA を使用して得られた違反検出の総数

分析対象となったこれら 2 つのソフトウェア開発プロジェクトにおいて、C 言語プロジェクトである映像サービスの通信プロトコルでは、CERT 拡張チェッカーグループによる違反検出数が Fortify SCA デフォルトのチェッカーグループによる違反検出数よりも多い結果となった（408 対 396）。逆に、C++言語プロジェクトである料金徴収システム関連の GUI アプリケーション では、C++言語用の CERT 拡張チェッカーグループによる違反検出数が 186 件であったのに対し、Fortify SCA デフォルトのチェッカーグループは 572 件であった。この結果はおそらく Fortify SCA 用に実装された C++言語用の CERT 拡張チェッカーの数が限られていたということで説明がつく。

表 5 に、2 つのソフトウェア開発プロジェクトに対し、Fortify SCA を使用して得られたすべての違反検出の中で、C/C++言語用の CERT 拡張チェッカーによる違反検出の分析結果を示す。なお、トゥルーポジティブ率は、次に示す公式により算出されている。

トゥルーポジティブ率 ＝ トゥルーポジティブの数 ÷（違反検出数 － 不明の数）

開発プロジェクト	言語	違反検出数	トゥルーポジティブ	フォールスポジティブ	不明	トゥルーポジティブ率
料金徴収システム関連の GUI アプリケーション	C++	186	125	61	0	67%
映像サービスの通信プロトコル	C	408	90	100	218	47%

表 5：拡張チェッカーにより Fortify SCA で検出された違反の分析結果

C++言語用 の CERT 拡張チェッカーではトゥルーポジティブ率が 67% であったのに対し、C 言語用の CERT 拡張チェッカーでは 47% のトゥルーポジティブ率であった。なお、本調査プロジェクトにおいて、C++ 言語用の CERT 拡張チェッカーでの違反検出レポートのすべてが分析・評価されたが、C 言語用の CERT 拡張チェッカーによる違反検出 レポートについては時間的制約により 408 件の内 190 件のみ分析・評価が実施された。表 5 の中の映像サービスの通信プロトコル開発プロジェクトのトゥルーポジティブ率 47% は、不明の数を除いた 190 件の違反検出結果の分析、評価認済みレポートに基づいた数値である。

4.1.1 CERT C++ セキュアコーディングスタンダードの有効性評価

Fortify SCA に実装された 23 個 の C++言語用 CERT 拡張チェッカーのうち 8 個（37.5%）のルールの違反が見つかった（表 3 を参照）。

表 6 に示すように、これらチェッカーのトゥルーポジティブ率にはばらつきがあった。

ルール・推奨事項	違反検出数	トゥルーポジティブ	フォールスポジティブ	不明	トゥルーポジティブ率
INT35-C	4	4	0	0	100%
RES32-C	8	8	0	0	100%
INT06-A	88	88	0	0	100%
INT31-C	12	7	5	0	58%
INT32-C	38	15	23	0	39%
DAN34-C	33	3	30	0	9%
INT33-C	1	0	1	0	0%
DCL30-C	2	0	2	0	0%

表 6：Fortify SCA に追加された C++言語用拡張チェッカーによる違反検出の分析結果

CERT C++ セキュアコーディングスタンダードの 1 つの推奨事項である INT06-A（「文字列トークンを整数に変換するときは `strtol()` または関連する関数を使用すること」）の 88 件の違反検出はトゥルーポジティブ率 100% であった。これは、同推奨事項用に実装されたチェッカーはこの推奨事項の違反を正確に検出できるということを示唆していると同時に、開発者がこれまではこの推奨されるコーディング手法を知らなかった可能性があるということも示している。検出結果の中で、DAN34-C（「無効なポインタを逆参照しないこと」）などのいくつかのルールの違反は、表 7 に示すようにかなり深刻な結果となる可能性がある。表 7 に示されたルール・推奨事項のリスク分析の基準については、付録 E を参照願う。

ルール・推奨事項	深刻度	可能性	修正コスト	優先度	レベル
DAN34-C	3（高）	3（高）	1（高）	P9	L2

表 7：DAN34-C に対するリスク分析値

残念ながら、CERT C++ セキュアコーディングスタンダードのルールである DAN34-C のチェッカーは 9% という低いトゥルーポジティブ率となった。この低い検出率の原因は、new オペレーションの失敗を捕捉するために例外処理コードブロック（try-block）が使用されていたが、こういったメモリ割り当てエラーが例外処理コードブロックによって捕捉されるケースを Fortify SCA が適切に認識することができないためである。このことは DAN34-C に関する違反検出結果の中で誤検出（フォールスポジティブ）として分析された 30 個 のすべてに当てはまり、この状況が適切に処理されれば DAN34-C のフォールスポジティブ率は大幅に低下する可能性があることを示唆している。

4.1.2 CERT C セキュアコーディングスタンダードの有効性評価

表 8 に、CERT C セキュアコーディングスタンダードのルールと推奨事項を実装したチェッカーを適用した Fortify SCA を使用して、 映像サービスの通信プロトコル開発プロジェクトのソースコードを分析した結果を示す。

ルール・推奨事項	違反検出数	トゥルーポジティブ	フォールスポジティブ	不明	トゥルーポジティブ率
ARR30-C	2	2	0	0	100%
INT32-C	75	47	28	0	63%
MEM35-C	40	24	16	0	60%
INT31-C	6	3	3	0	50%
INT30-C	44	14	30	0	32%
INT35-C	13	0	13	0	0%
MEM00-A	10	0	10	0	0%
POS31-C	1	0	0	1	
MSC30-C	1	0	0	1	
INT10-A	33	0	0	33	
INT14-A	14	0	0	14	
FIO45-C	1	0	0	1	
ENV30-C	1	0	0	1	
FIO33-C	3	0	0	3	
INT13-A	36	0	0	36	
TMP33-C	12	0	0	12	
STR03-A	33	0	0	33	
INT07-A	40	0	0	40	
MEM02-A	4	0	0	4	
INT06-A	39	0	0	39	

表 8 : Fortify SCA に追加された C 言語用拡張チェッカーによる違反検出の分析結果

時間的制約により、得られた違反検出結果の多くは完全には分析することができなかった。分析したルール・推奨事項のうち、MEM00-A、INT30-C、INT35-C が著しく悪い結果となった。MEM00-A のフォールスポジティブについては、同一関数内で、`malloc()` が呼び出されるが `free()` は呼び出されない場合 Fortify SCA は `free()` が呼び出されているかどうかを判断できないという制約が原因であることが分かっている。

INT30-C と INT35-C についての低いトゥルーポジティブ率の原因については明らかになっていない。

4.2 COMPASS / ROSE を用いたソースコード分析の結果とその評価

2 つのソフトウェア開発プロジェクトについて ROSE がレポートした違反検出の総数を表 9 に示す。結果は、プロジェクトおよびチェッカーグループごとに示されている。「デフォルト」と示されたチェッカーグループは、ROSE に標準装備のチェッカーによる違反検出を示す。「CERT 拡張」と示されたチェッカーグループは、、本調査プロジェクトの中で ROSE を拡張するために新たに開発された CERT C/C++ セキュアコーディングスタンダードのルール・推奨事項用に開発されたチェッカーによる違反検出を示す。

開発プロジェクト	言語	チェッカーグループ	違反検出数
料金徴収システム関連の GUI アプリケーション	C++	**CERT 拡張**	**200**
		デフォルト	1476
映像サービスの通信プロトコル	C	**CERT 拡張**	**7**
		デフォルト	70

表 9：ROSE を使用して得られた違反検出の総数

C++言語用、C 言語用共に CERT 拡張チェッカーによる違反検出数は、デフォルトの ROSE チェッカーが検出した数よりもはるかに少ない。また、C 言語用の CERT 拡張チェッカーが映像サービスの通信プロトコル 開発プロジェクトに対して検出した数は、C++言語用の CERT 拡張チェッカーが 料金徴収システム関連の GUI アプリケーション 開発プロジェクトで検出した数よりもはるかに少ない。これらの原因は、C++言語用の ROSE チェッカーは外部ヘッダファイルに対しての違反検出を多くレポートしたが、C 言語用の ROSE チェッカーはそうではなかったためである。ヘッダファイルは複数のプログラムファイルにインポートされることがよくあるため、多くのヘッダファイルレポートが重複して検出され報告された。この結果、1 つのヘッダファイルのレポートが、それを含むプログラムファイルごとに 1 つずつ検出されてしまうこととなった。

表 10 に、2 つのソフトウェア開発プロジェクトに対し、ROSE を使用して得られた総違反検出の中で、C/C++言語用の CERT 拡張チェッカーによる違反検出の分析結果を示す。なお、トゥルーポジティブ率は、違反検出数から不明の数を差し引いた数に対しての、トゥルーポジティブ数の割合である。

開発プロジェクト	言語	違反検出数	トゥルーポジティブ	フォールスポジティブ	不明	トゥルーポジティブ率
料金徴収システム関連の GUI アプリケーション	C++	200	19	51	130	27%
映像サービスの通信プロトコル	C	7	5	2	0	71%

表 10：拡張チェッカーにより ROSE で検出された違反の分析結果

表 11 では、ROSE を使用して違反を検出した CERT C/C++ セキュアコーディングスタンダードのルール・推奨事項と、それぞれの成功検出率（トゥルーポジティブ率）を一覧にし、より詳細な情報を示す。ROSE は、拡張実装された 15 個 の C++言語用の CERT 拡張チェッカーの内 6 個と、C 言語用に拡張実装された 12 個 の CERT 拡張チェッカーの内 3 個について違反を検出した。6 つの C++ 言語用の CERT 拡張チェッカーとはルール・推奨事項 ERR01-A、ERR02-A、OBJ32-C、OBJ00-A、OBJ03-A、RES35-C に基づくもので、C 言語用の CERT 拡張チェッカーは MSC33-C、ARR00-A、STR31-C に基づくものである。

言語	ルール・推奨事項	違反検出数	トゥルーポジティブ	フォールスポジティブ	不明	トゥルーポジティブ率
C++	OBJ32-C	23	9	0	14	100%
	ERR02-A	18	10	8	0	56%
	ERR01-A	43	0	43	0	0%
	OBJ00-A	7	0	0	7	
	OBJ03-A	72	0	0	72	
	RES35-C	37	0	0	37	
C	ARR00-A	3	3	0	0	100%
	MSC33-C	1	1	0	0	100%
	STR31-C	3	1	2	0	33%

表 11：ROSE に追加された C++、C 言語用拡張チェッカーによる違反検出の分析結果

C 言語プロジェクトの映像サービスの通信プロトコルの分析結果では、フォールスポジティブが 1 つの CERT 拡張チェッカー（STR31-C）にしかなく、良い結果を示している。ARR00-A（「型のサイズを知るために sizeof 演算子を使う場合は慎重に行うこと」と MSC33-C（「擬似乱数の生成に関数 `rand()` を使用しないこと」）の 2 つの CERT 拡張チェッカーにおいてはフォールスポジティブがなく、どちらもかなりはっきりとした傾向を示している。ARR00-A は不適切な場所での特定のコーディングイディオムの使用を検出しており、MSC33-C は `rand()` 関数の使用を見つけている。なお、ARR00-A は表 12 に示すように深刻な事態に繋がる可能性がある。表 12 に示されたルール・推奨事項のリスク分析の基準については、付録 E を参照願う。

ルール・推奨事項	深刻度	可能性	修正コスト	優先度	レベル
ARR00-A	3（高）	2（中）	3（低）	P9	L2

表 12：ARR00-A に対するリスク分析値

ROSE は多くの外部ヘッダファイルに対して、ルール・推奨事項への違反を検出した。これらのファイルはオペレーティングシステム、ROSE 自体、または 料金徴収システム関連の GUI アプリケーション開発プロジェクトで使用される外部ソフトウェアパッケージの一部であった。時間的制約のため、これらの違反は分析せず、「不明」として報告している。外部ヘッダファイルに関する ROSE の違反検出エラーを除外することが賢明かもしれないが、分析対象の開発プロジェクトのソースコードの一部であるヘッダファイルは決して除外してはならない。なぜなら、通常こういったヘッダファイルにはクラス定義、テンプレート定義、名前空間が含まれており、これらはコーディングルール違反の大いなる源だからである。ヘッダファイルレポートの大まかな解析については次のセクションに記述した。

外部ヘッダファイル以外の違反検出結果はかなり有望である。6 個の CERT 拡張チェッカーの内 3 個（OBJ32-C, ARR00-A, MSC33-C）には、違反検出のフォールスポジティブがなく、いくつかのトゥルーポジティブがあった。なお、1 つの CERT 拡張チェッカー（ERR01-A）が、全体で 53 個 のフォールスポジティブの内 43 個 を占めている。ERR01-A にはトゥルーポジティブとして分析された違反検出がなかったので、このルールを削除すれば、コストをかけずに全体の違反検出成功率を大幅に向上できるであろう。しかし、結果は、調査の整合性を保つために保持された。幸いにも、フォールスポジティブとなった違反検出もあった他のルール（ERR01-A, STR31-C）は、トゥルーポジティブとして分析される違反検出もあった。2 つの CERT 拡張チェッカー（ERR02-A と STR31-C）はフォールスポジティブ、トゥルーポジティブとして分析された違反検出があった。これらルール・推奨事項に関連する呼び出しがかなり少ないこともあり、数値上有望な成功検出率（トゥルーポジティブ率）を示している（ぞれぞれ 56% と 33%）。

ROSE による違反検出結果の分析とその評価作業は、検出数が少ないこともあり、処理しやすい作業であった。ROSE 用に開発された C++言語用の CERT 拡張チェッカーではトゥルーポジティブ率が 27% であったのに対し、C 言語用の CERT 拡張チェッカーでは 71% のトゥルーポジティブ率であった。C 言語用の CERT 拡張チェッカーによる 7 件すべての 違反検出が分析されたが、C++言語用の CERT 拡張チェッカーによる違反検出については時間的制約により 400 件のうち 70 件のみ違反検出結果の分析、評価を実施した。なお、C++言語用の CERT 拡張チェッカーにおける 27% という低いトゥルーポジティブ率は主に ERR01-A が原因である。ERR01-A だけで、C++言語プロジェクトである料金徴収システム関連の GUI アプリケーション開発プロジェクトについての全 51 個 のフォールスポジティブの内で 43 個が同チェッカーによる誤検出で占められている。

ERR01-A の高いフォールスポジティブ率の原因分析を実施した結果、ERR01-A および ERR02-A のフォールスポジティブの理由が判明した。これらの推奨事項は、`std::exception` から継承されたオブジェクト以外を throw 、catch すべきではないとしている。しかし、フォールスポジティブとして評価された違反検出は実際には、推奨事項に適合する形で標準 C++ インクルードファイルの中で提供された標準例外オブジェクトを throw、catch していたが、同推奨事項を実装した CERT 拡張チェッカーは、この例外オブジェクトを適切に認識することができなかった。今後この問題に対応するために ROSE と同チェッカーのソースコードを修正する予定であり、このようなフォールスポジティブは再発しないはずである。

なお、Fortify SCA と ROSE の両方が違反を検出した CERT C セキュアコーディングスタンダードのルールが 1 つ存在した。表 13 は、このルールに対する ROSE と Fortify SCA 両方による分析結果を示している。なお、表中に示されている、異なるルールの識別子 MSC30-C と MSC33-C は、実は同じルールであり、正しくは MSC30-C として統一されている点に注意願う。

開発プロジェクト	ツール	ルール	ファイル名	行番号	分析結果
映像サービス通信プロトコル	Fortify SCA	MSC30-C	File00039.c	44	不明（検知されたが未分析）
	ROSE	MSC33-C	File00039.c	44	トゥルーポジティブ

表 13：Fortify SCA および ROSE の双方で違反が検出されたルール

Fortify SCA と ROSE はどちらも、このルールの 1 つの違反を特定しており、どちらも同じファイルの同一行を特定している。ROSE による検出結果はトゥルーポジティブと見なされたが、Fortify SCA による検出結果は時間的制約のため分析されなかった。これは、Fortify SCA と ROSE が同じ違反を報告していると推測して差し支えない。

なお、ここで取り上げられたルール（MSC30-C と MSC33-C）は、`rand()` 関数を疑似乱数生成の関数として使うことに対して警告を出す。これは、同関数の使用の結果、乱数として適切な結果が得られないためである。同ルールへの違反検出のために必要なのは、たとえば `rand()` 関数呼び出しなどをソースコードの中で探すだけなので、これは容易に違反検出を行うことができるセキュアコーディングルールである。

4.2.1 ROSE による外部ヘッダに対する違反検出の解析

ROSE は C++ ヘッダファイルに関する違反検出を多数発行したため、これらのヘッダファイルへの違反検出について ROSE の簡単な解析を行った。図 2 に概要を示した小さなプログラムに対して、ROSE の診断ツールを実行することにより、解析を行った。このプログラムは Linux（Ubuntu 7.10）上で動作し、ヘッダファイルは ROSE（バージョン 0.9.1a）によって提供されるので、この解析作業においては実際にはコンパイラは使用されなかった。結果を表 14 に示す。

```
#include <locale>
#include <vector>

int main() {
  return 0;
}
```

図 2：簡易ヘッダ診断プログラム

ルール・推奨事項	ファイル名	行	違反検知メッセージ
OBJ00-A	bits/locale_classes.h	105	は public データである。
OBJ32-C	bits/locale_classes.h	524	_Impl は非明示的単一引数コンストラクタである。
OBJ00-A	bits/ios_base.h	256	は public データである。
OBJ00-A	bits/ios_base.h	469	は public データである。
OBJ00-A	bits/ios_base.h	498	は public データである。
OBJ00-A	bits/locale_facets.h	697	は public データである。
OBJ03-A	bits/locale_facets.h	1003	は1020 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1020	は1003 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1036	は 1053 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1053	は 1036 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1073	は1096 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1096	は 1073 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1122	は 1148 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1148	は 1122 行目の仮想関数をオーバーロードする
OBJ00-A	bits/locale_facets.h	1236	は public データである。
OBJ03-A	bits/locale_facets.h	1280	は 1299 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1299	は 1280 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1352	は 1369 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1369	は 1352 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1385	は 1402 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1402	は 1385 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1422	は1444 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1444	は 1422 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1467	は1493 行目の仮想関数をオーバーロードする
OBJ03-A	bits/locale_facets.h	1493	は 1467 行目の仮想関数をオーバーロードする
OBJ00-A	bits/codecvt.h	346	は public データである。
OBJ00-A	bits/codecvt.h	404	は public データである。
OBJ00-A	bits/stl_bvector.h	71	は public データである。
OBJ00-A	bits/stl_bvector.h	112	は public データである。
OBJ32-C	bits/stl_bvector.h	282	_Bit_const_iterator は非明示的単一引数コンストラクタである。
RES35-C	pthread.h	520	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	pthread.h	520	ポインタデータメンバを持つクラスは、コピーコンストラクタ、コピー代入、デストラクタを定義すべきと想定される。
RES35-C	exception	54	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	exception	66	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	new	55	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	bits/ios_base.h	207	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	bits/ios_base.h	466	ポインタデータメンバを持つクラスは、コピーコンストラクタ、コピー代入、デストラクタを定義すべきと想定される。
RES35-C	bits/ios_base.h	496	ポインタデータメンバを持つクラスは、コピーコンストラクタ、コピー代入、デストラクタを定義すべきと想定される。
RES35-C	bits/ios_base.h	530	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	typeinfo	139	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	typeinfo	149	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	bits/stl_bvector.h	69	コピーコンストラクタ、コピー代入、デストラクタのいずれかが宣言されるなら、3つすべてを宣言せよ。
RES35-C	bits/stl_bvector.h	69	ポインタデータメンバを持つクラスは、コピーコンストラクタ、コピー代入、デストラクタを定義すべきと想定される。

表 14：ROSE が検出した標準ヘッダファイルにおける違反結果

CERT C++ セキュアコーディングスタンダードの推奨事項の一つである OBJ00-A は、クラスメンバデータの public 宣言を禁止している。C++ においては、1 つの `struct` は、単にすべてのメンバがパブリック宣言されたクラスである。したがって、推奨事項 OBJ00-A を実装した CERT 拡張チェッカーはすべての `struct` のメンバ宣言を検出する。

同じく、推奨事項である OBJ03-A では、派生クラスが継承された仮想関数の一部のみをオーバーロードした結果その他の継承された仮想関数を隠蔽してしまう恐れがあるため、仮想関数をオーバーロードすることを避けるべきであるとしている。`locale_facets.h` は文字型のクラスとテンプレートを定義しており、幾つかのクラスが `do_widen()`、`do_narrow()`、`do_toupper()`、`do_tolower()` などのメソッドを個々クラスに該当する文字型に対して提供している。これらクラスの各メソッドは仮想関数をオーバーロードしている（各クラスは 1 文字を操作するメソッドと、特定の文字範囲を操作するメソッドをそれぞれ 1 つ定義している）。この場合、このヘッダは明らかに OBJ03-A で定義された推奨事項に違反している。こういったケースは、同推奨事項が間違っているか、あるいはヘッダが間違っている可能性があるが、OBJ03-A の違反検出を行うためのチェッカーは正しく違反を検出していることが分かる。

推奨事項ではなくルールとして分類されている OBJ32-C では、すべての単一引数のコンストラクタを `explicit` 宣言することを定めている。これにより、開発者は明示的に指定したコンストラクタを使用するか、型キャストを行うことによって型変換を強制され、想定しない型変換を防止できる。OBJ32-C に関する違反検出結果はベクタ型の内部の explicit 宣言されていない単一引数コンストラクタ宣言に対しての検出であった。この結果から、OBJ32-C は CERT C++ セキュアコーディングスタンダードのルールではなく推奨事項として再分類すべきだということを示している。

最後に RES35-C は 2 つの要素から構成されている。第 1 に、クラスがコピーコンストラクタ、代入演算子、またはデストラクタを宣言する場合、このクラスは 3 つすべての対応するメソッドを宣言すべきである。第 2 に、クラスがオブジェクトへのポインタを持っている場合にも 、これら 3 つすべてのメソッドを宣言すべきである。このことで、あるオブジェクトがそのオブジェクトが参照する他のオブジェクトに対して管理責務を持つ必要性が明確になる。

`<pthread>` 内の RES35-C に対する違反検出結果はすべてポインタメンバを持つクラスについての問題を指摘している。それ以外の RES35-C の違反検出結果はすべて仮想メンバ関数と対応する仮想デストラクタを持つクラスに対して、代入演算子やコピーコンストラクタが不足している点を指摘している。

このようなすべての場合に、これらチェッカーは対応したルール・推奨事項に記述されたとおり忠実に違反検出を行った。しかしながら、単純にルール・推奨事項が特定の例外的なケースを考慮できていない（たとえば、仮想デストラクタ）、ルール・推奨事項が過剰に厳しく解釈されている（たとえば、`structs` を許可しないこと）、または、特別な状況においてセキュリティを損なわずにあえてルールを破る場合もある（たとえば、explicit 宣言なしの 単一引数コンストラクタ）などのルール・推奨事項の定義そのものの課題が明らかとなった。最終的には、これらすべての外部ヘッダに対して違反検出されたルール・推奨事項の深刻度が低いとみなされた。なぜならこれらのルール・推奨事項への違反は、深刻な結果を引き起こすことが少なく、攻撃可能な脆弱性を引き起こし難いためである。また、これらはすべて一般的な C++ スタイルガイドラインに準拠している。

結論として、実装されたチェッカーは対応するルール・推奨事項に忠実に機能しており、ヘッダファイルのルール違反を実際に検出している。しかし、これらの違反は必ずしも攻撃を引き起こすセキュリティの脆弱性ではなく、むしろルールや推奨事項に対する有効な例外を示すものである。最終的には、必要に応じて例外を追加することを含めルールと推奨事項の定義を見直した上で、それに応じた例外を適切に処理するようにチェッカーを修正する必要がある。

5 まとめ

ソースコード解析ツールは、多くの場合ソフトウェア製品の脆弱性の一因となるソフトウェアの欠陥を見つけ、除去するために効果的に使用できる。拡張チェッカーを追加しないバージョンの Fortify SCA と ROSE は、商用目的で開発されたソフトウェアのソースコードから、ソフトウェアの脆弱性に結びつく可能性のあるソフトウェア欠陥（修正されずに残っている）を発見することができる。しかし、CERT C/C++ セキュアコーディングスタンダードへの適合状況を確かめるための追加実装されたチェッカーの開発とその適用は、両ツールにとって有用であるということが、今回の調査によって示された。

Fortify SCA と ROSE はどちらも構文解析（および一部の意味解析）ができるため、これらを拡張することで、ソースコードの CERT C/C++ セキュアコーディングスタンダードのルールおよび推奨事項への違反を検出することができる。ROSE は構文解析ツリーを正規化しないため、構文解析においてやや優れている。結果として、ROSE は、Fortify SCA ではチェックできないシグナル処理などの分野の違反を発見できる。一方、Fortify SCA は多くのデフォルトチェッカーが標準装備されている点と、商用ツールとして ROSE よりも製品として洗練されている点で優れている。

ROSE も Fortify SCA も静的解析のみを実行する。この制約のため、Fortify SCA や ROSE が検証できないコードを作成することが可能である。事実、実際にコードを実行しなければ簡単には検証できないコードを作成するのは簡単である。これにより、静的解析が動的解析に優る可能性のある利点がすべて損なわれる。CERT セキュアコーディングスタンダードに適合していることを完全に確認するのは不可能なので、本調査プロジェクトでは、静的解析を使って検出されやすいルールと推奨事項への違反を発見することに焦点を絞った。

ROSE も Fortify SCA もプリプロセッサ指示文を処理するセキュアコーディングルールへの適合を確認することはできない。これは驚くことではない。なぜなら、指示文は別のプリプロセッサによって処理されるので、ほとんどの C コンパイラもこの問題を発見することができないためである。現在 ROSE にマクロを認識する機能を追加する研究プロジェクトが行われているが、まだ広範囲な使用の準備は整っていない。Fortify SCA には、構文解析ツリーを単純化する方法において、別の制約がある。Fortify SCA は、構文解析ツリーを単純化する過程において、一部の型変換、 および `sizeof()` 演算子やシグナル処理を除去する。この結果、Fortify SCA は `sizeof()` やシグナルの使用に関するルールや推奨事項の違反を発見するためには使用できない。

ルール・推奨事項への違反を検出するチェッカーを記述しテストする作業が、滅多に発生しない事例やルール・推奨事項に定義すべき例外の見直しを必要とし、ルール・推奨事項自体を洗練することに役立つ。特に、ルール・推奨事項へ違反することが有効な意味を持ち、この重要度が発生する可能性があるコストを上回ると判断された場合は、特別な条件下で（explicit 宣言なしの単一引数コンストラクタなど）C++ 推奨事項の一部を無視する可能性がある。このようなケースは CERT C++ セキュアコーディングスタンダードで許可される例外として列挙される必要がある。

最後に、多くのセキュアコーディングスタンダードのルールや推奨事項は、まだ Fortify SCA にも ROSE にも実装されていない。さらに、C および C++ セキュアコーディングスタンダードのルール・推奨事項そのものの改良に伴い、対応するチェッカーを改良する必要がある。

全体として、拡張された 2 つのソースコード解析ツールの使用を通して、2 つの評価対象ソフトウェア開発プロジェクトで首尾よく多くのソフトウェアの欠陥を検出し、CERT C/C++ セキュアコーディングスタンダードとソースコード解析ツールのどちらもソフトウェアの品質向上において有効であることが証明された。このプロジェクトではまた、CERT C/C++ セキュアコーディングスタンダードと評価された 2 つのソースコード解析ツールの両方をさらに改善する方法も特定することができた。

付録 A　Fortify SCA 向け CERT C セキュアコーディングスタンダード拡張チェッカー実装のための補足情報

以下に記述された情報は、解析プロセスのアーティファクトであり、本報告書の本文に提示した解析結果を補足するものである。この情報は完全でもなく、また、決定的なものでもないため、注意深く使用願う。

ルール・推奨事項	深刻度	進捗	説明	備考
PRE00-A	低	なし	マクロではなくインライン関数を使用すること。	Fortify は前処理の完了後にコードを解析する。
PRE01-A	低	なし	マクロ内の変数名を括弧で囲むこと。	Fortify は前処理の完了後にコードを解析する。
PRE02-A	低	なし	マクロの展開は、関数に似たマクロの場合は括弧で囲むこと。	Fortify は前処理の完了後にコードを解析する。
PRE03-A	低	なし	関数を呼び出そうとする場合はマクロの呼び出しは避けること。	Fortify は前処理の完了後にコードを解析する。
PRE05-A	低	なし	演算子を含むマクロ定義はすべて括弧で囲むこと。	Fortify はこれを検出できず、コードは前処理のあとに解析される。
PRE30-C	低	なし	連結を通じてユニバーサル文字の名前を作成しないこと。	Fortify は前処理の完了後にコードを解析する。
PRE31-C	低	なし	代入、インクリメント、デクリメント、または関数呼び出しを含んだ引数を使って安全でないマクロを呼び出さないこと。	Fortify は前処理の完了後にコードを解析する。
DCL01-A	低	なし	サブスコープ内で変数名を再利用しないこと。	Fortify は異なる 2 つのスコープの内容は比較できない。
DCL02-A	低	なし	視覚的に明確に区別できる識別子を使うこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
DCL03-A	低	なし	右端の宣言指定子には const を置くこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
DCL04-A	低	なし	1 つの宣言に対して複数の変数を宣言する際は慎重に行うこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。

DCL05-A	低	なし	コードを読みやすくするために型定義を使用すること。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
DCL06-A	低	なし	プログラムロジックでリテラル値を表現する場合は意味のあるシンボリック定数を使うこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
DCL07-A	低		関数宣言子には型情報を含めること。	これらのコードサンプルはいずれもコンパイル用に入手できなかった。これらの推奨事項はコンパイラによって解決されている可能性がある。
DCL08-A	中	なし	互換性のある型を使って関数ポインタを宣言すること。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
DCL09-A	低	なし	`errno_t` 型の `errno` を返す関数を宣言すること。	Fortify は関数の戻り値を検出できない。
DCL10-A	中	なし	可変個引数の関数の使用は慎重に行うこと。	`insert (loc++, ..)` は次の 2 つの文に変換されるためこれは不可能。`loc_0 = loc++` と `insert(loc_0, ...)`。
DCL30-C	高	FORTIFY で一部	適切な記憶域期間でオブジェクトを宣言すること。	Fortify は、配列を宣言してから関数内でのその配列へのポインタを返すケースを捕捉する。Fortify ではそれ以上は不可能。
DCL32-C	低	なし	識別子が確実に一意になるようにすること。	Fortify にはこの問題に適切に対処するメカニズムがない。
DCL33-C	中	なし	関数引数内のソースと宛先のポインタが restrict で修飾されている場合、それらが重複したオブジェクトを指さないようにすること。	
DCL34-C	中	なし	キャッシュできないデータには volatile を使う。	評価担当者は Fortify ではこれはできないと考える。
DCL36-C	低	なし	リンクの分類が異なる識別子を使わないこと。	構造アナライザではこれは発見できない。現在、Fortify フロントエンドは複数定義を解決しようと取り組んでおり、アナライザに一貫したビューを提供しているため、構造ルールは複数定義を検出していない。
EXP00-A	低	なし	演算の優先順位指定には括弧を使うこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
EXP01-A	高	なし	型のサイズを知るためにポインタの `sizeof` を取得しないこと。	`sizeof` は Fortify がコードを解析できるようになる前に処理される。
EXP02-A	低	なし	論理 AND および論理 OR 演算	`&&` を検出できない。

			子の第 2 オペランドに副次的効果を含めないこと。	Fortify フロントエンドは、
				`if ( a && b) ... を if (a)`
				`{ if (b) ... }`
				に変換し、 `'(i++) == max'` を `'tmp = i; i = i + 1; tmp == max'` に変換する。
				このケースに対処するために、Fortify はこの変換を反転させる必要があるが、現時点ではできない。

EXP03-A	中	なし	構造体のサイズはそのメンバのサイズの合計であると想定しないこと。	`sizeof` は Fortify がコードを分析できるようになる前に処理される。
EXP04-A	中	なし	構造体どうしでバイト単位の比較をしないこと。	`memcmp()` を検出するセマンティックルールを作成できるが、その `fn` に渡される構造体は検出できない。
EXP05-A	低	なし	const 修飾子をキャストによって除去しないこと。	const を検出できない。
EXP06-A	低	なし	`sizeof` 演算子に対するオペランドに副次的効果を含めないこと。	`sizeof` は Fortify がコードを分析できるようになる前に処理される。
EXP08-A	高	なし	ポインタ算術演算は正しく使うこと。	`sizeof` は Fortify がコードを分析できるようになる前に処理される。
EXP09-A	高	なし	型または変数のサイズを知るには `sizeof` を使うこと。	`sizeof` は Fortify がコードを分析できるようになる前に処理される。
EXP30-C	中	一部	シーケンスポイント間の評価の順番に依存しないこと。	
EXP31-C	低	なし	定数値は変更しないこと。	
EXP32-C	低	なし	非 `volatile` の参照を通じて volatile オブジェクトにアクセスしないこと。	
EXP33-C	高	FORTIFY 一部	初期化されていない変数を参照しないこと。	これはサンプルコードを捕捉するが、常に初期化を認識するわけではない。別の関数で初期化が行われている場合は認識されない。ポインタが使われていると予期しない動作を引き起こす。多数のフォールスポジティブがある。Fortify は、「low :Uninitialized

				Variable :Controlflow」として捕捉する。
EXP33-C	高	FORTIFY 一部	初期化されていない変数を参照しないこと。	これはサンプルコードを捕捉するが、常に初期化を認識するわけではない。別の関数で初期化が行われている場合は認識されない。ポインタが使われていると予期しない動作を引き起こす。多数のフォールスポジティブがある。Fortify は、「low :Uninitialized Variable :Controlflow」として捕捉する。
EXP34-C	高	FORTIFY	ポインタを逆参照する前にその有効性を確認すること。	
EXP35-C	低	なし	次のシーケンスポイントよりもあとの関数呼び出しの結果へのアクセスや変更を行わないこと。	
EXP36-C	低	なし	アラインメントが異なるオブジェクトや型へのポインタ間でキャストしないこと。	
EXP37-C	低	なし	正しい引数で関数を呼び出すこと。	この構造ルール言語は、現時点では我々がサポートしていない全称量化子 `forall` を必要とするため、関数呼び出し違反の検出に必要な述語をサポートしていない。
INT01-A	中	一部	オブジェクトのサイズを表すすべての整数値に `size_t` を使うこと。	これは INT32-C のルールによって部分的にカバーされているが、Fortify は `size_t` 型を検出できず、`size_t` を unsigned long として認識する。
INT05-A	中	完了	文字データの入力とデータの変換を行う関数が、考えられるあらゆる入力に対応できない場合は、これらの関数使わないこと。	これは FIO33-C でカバーされている。
INT06-A	中	完了	文字列トークンを整数に変換するときは strtol() を使うこと。	atoi、atol、atoll に文字列が渡された場合にこれらを検出する構造ルールを作成した。
INT07-A	中	完了	文字型に対して、符号付きまたは符号なしを明示的に指定すること。	int が char（unsigned char や `signed char` ではなく）に割り当てられたことや、`char` を使って演算が行われたことをチェックするルールを作成した。
INT09-A	低	なし	列挙定数が一意の値にマップすることを確実にする。	列挙型を検出できない。
INT10-A	低	完了	% 演算子の剰余の結果値の符号を仮定しないこと。	% を検出する構造ルールを作成した。
INT13-A	高	完了	右シフト演算が論理シフトまたは算術シフトとして実装されている	右シフト演算が実行されたことを検出する構造ルールの作成が

			と想定しないこと。	可能。
INT14-A	中	一部	ビットマップと数値型を区別すること。	ビット処理演算と同じ行で算術演算が実行されているかを照合する構造ルールを作成できる。正数の演算と負数の演算の違いは区別できない。
INT30-C	低	一部	符号が疑問視される結果を対象に特定の演算を行わないこと。	現在、型のサイズを判定する方法はないが、SCA 5.0 では、構造アナライザの Type オブジェクトには型が占めるバイト数を示す storageSize プロパティが追加される予定である。現時点では、MyStruct* に Type オブジェクトがある場合、MyStruct* のサイズは調べられるが、「ポインタを削除」するだけでは MyStruct のサイズは判明しない。サイズを調べるには、Type オブジェクトをツリー構造にするような拡張が必要である。
INT31-C	高	完了	整数変換によってデータの消失や解釈間違いが発生しないことを保証すること。	代入式の左辺の変数をチェックせずに型変換を見つける構造ルールを作成できる。
INT32-C	高	完了	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こと。	対象となる演算が実行され、if 文がないかを判定する構造ルールを作成できる。 追加のテストで多くのフォールスポジティブが明らかになり、そのほとんどは for ループに関連していた。for ループ文の一部でない場合にのみ検出するチェックを追加した。これがこのようなフォールスポジティブを解決する正しい方法かどうかは不明。
INT33-C	低	完了	除算演算およびモジュロ演算がゼロ除算エラーを引き起こさないことを保証すること。	INT32-C に似たルールを作成した。
INT35-C	高	完了	大きなサイズの整数に対して比較や代入を行う場合は事前に整数をアップキャストすること。	
INT36-C	高	完了	負のビット数、またはオペランドにあるよりも多くのビットをシフトしないこと。	INT32-C のルールによってカバーされている。
INT37-C	低	完了	文字処理関数への引数は、`unsigned char` として表現できるようにすること。	ctype.h からの fn を処理する `char` に `unsigned char` 以外のものが渡されたことを検出する構造ルールを記述した。
FLP30-C	低	完了	浮動小数点値の比較時には精度を考慮すること。	If 文に f == g の形式で比較される 2 つの浮動小数点数が含まれていることを検出する構造ルールを作成した。

FLP31-C	低	なし	複素数値を使った実値を期待する関数を呼び出さないこと。	Fortify はこれらの関数のいずれも検出しない。
FLP32-C	中	完了	数学関数におけるドメインエラーを防止すること。	これを処理する制御フローを作成した。
FLP33-C	低	一部	浮動小数点演算では整数を浮動小数点数に変換すること。	型が double または float の変数の代入文に、double や float の結果にならない演算が含まれていることを検出する構造ルールを作成した。Fortify は、作成したルールでは、1 つ目の適合コードによる解決法を間違って検出する。
FLP34-C	低	完了	縮小変換後の浮動小数点値が確実に範囲内に収まるようにすること。	double 型または long double 型が float に縮小変換される場合や、大きな変数に、それよりも小さい変数では格納できない値が含まれていないかをチェックする if ブロックがないことを検出する構造ルールを作成した。
ARR00-A	高	なし	配列のサイズを知るために `sizeof` 演算子を使う場合は慎重に行うこと。	`sizeof` は Fortify がコードを分析できるようになる前に処理される。
ARR30-C	高	一部	配列のインデックスが確実に適切な範囲内にあるようにすること。	
ARR31-C	高	なし	すべてのソースファイルにわたり一貫性のある配列表記法を使用すること。	構造アナライザではこれは発見できない。現在、Fortify フロントエンドは複数定義を解決しようと取り組んでおり、アナライザに一貫したビューを提供しているため、構造ルールは複数定義を検出していない。DCL36-C と同じ理由でこれを実施できない。
ARR32-C	高	なし	可変長配列へのサイズ引数は、適切な範囲内であること。	配列が動的に割り当てられ、値が適切にチェックされていないことを検出する構造ルールを作成した。Fortify では単一関数の適用範囲外を認識できないため、これはサンプルの適合コードを検出した。 このルールは多数のフォールスポジティブを生成する。Fortify は（ほかの種類のポインタの参照とは異なり）配列を明示的に検出できず、このルールは、ルールに分類されない多くのものを検出する。
ARR33-C	高	FORTIFY	コピーは必ず十分なサイズの記憶領域に対して行われるようにすること。	NCCE でコードを捕捉する。
ARR34-C	高	なし	式内の配列の型は必ず互換性があるようにすること。	Fortify は配列へのアクセス/代入を検出せず、サンプルコード内の型の違いも区別しない。

STR02-A	中	FORTIFY	複雑なサブシステムに渡すデータをサニタイズすること。	Fortify はサンプルコードを次のように検出する。「\[1E605754626A177B9721905D023B495E :medium :Command Injection :semantic \]」
STR03-A	低	完了	不注意で NULL 終端バイト文字列を切り捨てないこと。	`strncpy`、 `strncat`、 `fgets`、 `snprintf` が呼び出され、結果がテストなしで使われたことを検出する制御フロールールを作成した。
STR05-A	低	なし	文字列リテラルが const によって修飾されるようにすること。	Fortify は `char` と `char const` の違いを区別できない。
STR06-A	低	FORTIFY	`strtok()` が文字列引数を変更しないと想定しないこと。	Fortify はこれを \[CDFD0C4C211178014C47940B7C19EA30 :medium :Missing Check against Null :controlflow \] と \[2F6C99890155DBEB91367CA22A5D7E74 :low :Obsolete :semantic \] によって捕捉する。
STR07-A	低	一部	文字データにはプレーンな `char` を使うこと。	“” タグがルールに追加された。`unsigned char` に文字列リテラルが割り当てられていることを検出する構造ルールを作成した。Fortify は `char` と `signed char` の違いを区別できないため、部分的な修正のみ。
STR30-C	低	なし	文字列リテラルを変更しようとしないこと。	現時点では Fortify では不可能。
STR31-C	高	FORTIFY	文字列の記憶域に、文字データと NULL 終端文字用に確実に十分な領域があるようにすること。	Fortify は `strcpy` と `getenv` のサンプルコードに次のように検出する。「high ：Buffer Overflow :Dataflow」。`sizeof` が使用されているために、Fortify が対応できないサンプルコードの 1 つである。
STR32-C	高	なし	すべてのバイト文字列を確実に NULL で終了させること。	
STR33-C	高	一部	ワイド文字の文字列を正しくサイズ指定すること。	
STR34-C	中	完了	文字をより大きな整数サイズに変換する場合は事前に unsigned 型に変換すること	int または long に `char` が割り当てられていることを検出する構文ルールを作成した。
MEM00-A	高	完了	メモリの割り当てと解放は、同一モジュールの同一抽象レベルで行うこと。	単に malloc のみ、または単に free のみの fn を捕捉する制御フロールールを作成した。
MEM02-	低	完了	`malloc()` からの戻り値は型変換	`malloc()` 型の戻り値が代入される型と一致しない場合に、変

A			しないこと。	数の代入を検出する構造ルールを作成した。
MEM30-C	高	FORTIFY	解放したメモリにアクセスしないこと。	
MEM31-C	高	FORTIFY	動的に割り当てられたメモリは必ず一度解放すること。	
MEM32-C	低	FORTIFY	重大なメモリ割り当てエラーを検出して対処すること。	
MEM33-C	低	なし	動的にサイズ指定される構造体には柔軟性のある配列メンバを使うこと。	Fortify は構造体の内側の配列宣言を検出できない。
MEM34-C	低	一部	動的に割り当てられたメモリのみ解放すること。	
MEM35-C	高	一部	オブジェクトに十分なメモリを割り当てること。	`malloc()`、`calloc()`、`realloc()` が呼び出されて `multsize_t` が呼び出されていない場合や、`memcpy()` が `size_t` 以外の長さ値で呼び出されたことを検出する構造ルールを作成した。Fortify は `sizeof` を検出できないため、3番目の不適合のサンプルを捕捉できない。
FIO30-C	高	FORTIFY	書式文字列からユーザ入力を除外すること。	
FIO31-C	中	なし	同じファイルを同時に複数回開かないこと。	Fortify はこれを検出できない。
FIO32-C	中	FORTIFY	ファイル操作エラーを検出して対処すること。	
FIO33-C	低	完了	未定義の動作を引き起こす入力/出力のエラーを検出して処理すること。	
FIO34-C	高	完了	文字入出力関数の戻り値の取得には int を使用すること。	
FIO35-C	高	完了	end-of-file およびファイルエラーの検出には `feof()` および `ferror()` を使用すること。	
FIO36-C	高	なし	`fgets()`の使用時に改行文字が読み取られると想定しないこと。	
FIO37-C	高	FORTIFY	文字データが読み取られたと想定しないこと。	
FIO38-C	低	一部	入力と出力に FILE オブジェクトのコピーを使用しないこと。	ポインタの問題はフォールスネガティブにつながる。
FIO39-C	中	完了	ストリームへの出力の直後にそのストリームの読み取りを行わないこと。	

FIO40-C	低	一部	`fgets()` が失敗したときは文字列をリセットすること。	
FIO41-C	中	なし	`getc()` または `putc()` に副次的効果のあるパラメータを指定して呼び出さないこと。	
FIO42-C	中	完了	ファイルは必要がなくなったら必ず正しくクローズすること。	
FIO43-C	高	FORTIFY では一部	制限のないソースから固定長の配列へデータをコピーしないこと。	Fortify は `gets()` と `scanf()` を捕捉するが、`getchar()` サンプルは捕捉しない。`getchar()` サンプルの解決方法はまだ不明。
FIO44-C	中	完了	`fsetpos()` では `fgetpos()` から返された値のみを使用すること。	
FIO45-C	中	なし	ファイルストリームを再オープンしないこと。	制御フロールールを試みたが、Fortify が対処できるよりも大きなスコープが必要であるためできなかった。
TMP30-C	高	なし	一時ファイルは一意かつ予測できないファイル名で作成すること。	
TMP32-C	高	FORTIFY	一時ファイルは排他的アクセス権で開くこと。	Fortify はこれを、深刻度の低いセマンティックルール「Insecure Temporary File（安全でない一時ファイル）」で捕捉する。しかしこれは、我々がアドバイスした適合した解決法（`mkstemp()`）も「Insecure Temporary File」で捕捉する。
TMP33-C	中	完了	プログラムの終了前に一時ファイルを削除すること。	`tmpfile_s()` や `mkstemp()` ではなく `tmpfile()`、`fopen()`、`mktemp()` などが呼び出されたことを検出する制御フロールールを作成した。
ENV30-C	低	完了	`getenv()` から返された文字列は変更しないこと。	完了を確認する。
ENV31-C	低	なし	環境ポインタを無効にする可能性のある操作のあとは、そのポインタに依存しないこと。	
ENV32-C	低	なし	`exit()` 関数を複数回呼び出さないこと。	
ENV33-C	低	なし	`longjmp()` 関数を呼び出して `atexit()` によって登録された関数の呼び出しを終了しないこと。	Fortify は 2 つの異なるスコープの内容を比較できないため、これを警告できない。
SIG30-C	高	なし	シグナルハンドラ内では async-	Fortify ではシグナルハンドラを

			safe 関数のみを呼び出すこと。	検出できない。
SIG31-C	高	なし	シグナルハンドラ内の共有オブジェクトにアクセスまたは変更を行わないこと。	Fortify ではシグナルハンドラを検出できない。
SIG32-C	高	なし	シグナルハンドラ内から `longjmp()` を呼び出さないこと。	Fortify ではシグナルハンドラを検出できない。
SIG33-C	低	なし	`raise()` 関数を再帰的に呼び出さないこと。	Fortify ではシグナルハンドラを検出できない。
MSC30-C	低	完了	rand 関数を使用しないこと。	
MSC31-C	低	なし	戻り値が確実に適切な型と比較されるようにすること。	Fortify は `time_t` と long や、`size_t` と unsigned long を区別できない。
POS30-C	低	なし	`readlink()` 関数を正しく使用すること。	`sizeof` は Fortify がコードを分析できるようになる前に処理される。
POS31-C	中	完了	別のスレッドのミューテックスのロック解除や破壊を行わないこと。	関数がロックを取得する前にそれを破壊した場合に検出する制御フロールールを作成した。
POS32-C	中	なし	マルチスレッド環境でビットフィールドを使用する場合はミューテックスを含めること。	Fortify はこれを検出できない。ここではスコープの問題と前処理の問題がある。
POS33-C	低	FORTIFY	`vfork()` は使用しないこと。	
POS34-C	高	完了	引数に自動変数を使って `putenv()` を呼び出さないこと。	静的でない変数を使って `putenv()` が呼び出されたことを検出する Fortify の構造ルールを作成した。

付録 B　Fortify SCA 向け CERT C++ セキュアコーディングスタンダード拡張チェッカー実装のための補足情報

以下に記述された情報は、解析プロセスのアーティファクトであり、本報告書の本文に提示した解析結果を補足するものである。この情報は完全でもなく、また、決定的なものでもないため、注意深く使用願う。

ルール・推奨事項	深刻度	進捗	説明	備考
PRE31-C	低	なし	マクロではなくインライン関数を使用すること。	Fortify は前処理の完了後にコードを分析する。
PRE32-C	低	なし	マクロ内の変数名を括弧で囲むこと。	Fortify は前処理の完了後にコードを解析する。
PRE33-C	低	なし	マクロの展開は必ず括弧で囲むこと。	Fortify は前処理の完了後にコードを解析する。
DCL01-A	低	なし	サブスコープ内で変数名を再利用しないこと。	Fortify はスコープの問題は解決できない。
DCL02-A		なし	視覚的に明確に区別できる識別子を使うこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
DCL03-A		なし	右端の宣言指定子には const を置くこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
DCL04-A	低	なし	1 つの宣言に対して複数の変数を宣言しないこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。
DCL30-C	低	完了	実装用に予約された名前を使わないこと。	2 文字のアンダースコアによる変数、または 1 文字のアンダースコアと大文字 1 字による変数を捕捉する構造ルールを作成した。
DCL31-C	低	FORTIFY	自己初期化を避けること。	Fortify はこれを「Poor Style :Redundant Initialization :Structural」として検出する。
EXP06-A		なし	演算の優先順位指定には括弧を使うこと。	これは単純な `grep` で実行できる。Fortify にはこれを実行する手段が組み込まれていないようである。

EXP07-A	低	なし	`sizeof` 演算子へのオペランドに副次的効果を含めないこと。	`sizeof` は Fortify がコードを分析できるようになる前に処理される。
EXP30-C	低	なし	`const` をキャストによって除去しないこと。	Fortify は変数が `const` かどうかを区別できない。
EXP31-C	低	なし	定数値は変更しないこと。	C のルールと同じ。
EXP32-C	低	なし	非 `volatile` の参照を通じて volatile オブジェクトにアクセスしないこと。	C のルールと同じ。
EXP33-C	低	FORTIFY では一部	初期化されていない変数を参照しないこと。	C のルールと同じ。これはサンプルコードを捕捉するが、常に初期化を認識するわけではない。別の関数で初期化が行われている場合は認識されない。ポインタが使われていると予期しない動作を引き起こす。多数のフォールスポジティブがある。Fortify は、「low :Uninitialized Variable :Controlflow」として捕捉する。
EXP34-C	中	なし	シーケンスポイント間の評価の順番に依存しないこと。	EXP30-C C のルールと同じ。シーケンスポイントを識別する必要があるが、Fortify で実装するのはおそらく難しい。
EXP35-C	高	完了	シフト演算の右辺のオペランドが確実に範囲内にあるようにすること。	制御フロールールでこれが可能かどうかは不明。 これは INT31-C の構造ルールで捕捉される。2 つのルールは関連しており 1 つのルールで両方を十分に捕捉できると思われるため、INT31-C に捕捉を任せ、これを「完了」とマークする。
EXP36-C	中	一部	不完全クラスへのポインタを型変換および削除しないこと。	型 \[UnknownType\] の変数に \[UnknownType\] 以外が代入されたことを検出する構造ルールを記述した。Fortify では型変換を検出する方法はない。クラス宣言の内側では検出しない。
EXP37-C	低	一部	アサーションでの副次的効果を避けること。	main 以外の関数内で `assert( index+\+ == 0 );` または `assert( index == c.size() );` を捕捉する構造ルールを作成できたが、（この例の）index や c が、それを囲んでいる関数のパラメータであることを確実にすることはできなかった。

INT01-A		一部	オブジェクトのサイズを表すすべての整数値に `size_t` を使うこと。	これは INT32-C のルールによって部分的にカバーされているが、Fortify は `size_t` 型を検出できず、`size_t` を unsigned long として認識する。
INT05-A		完了	書式付きの入力を使って整数値を入力しないこと。	これは FIO33-C でカバーされている。
INT06-A		完了	文字列トークンを整数に変換するときは `strtol()` を使うこと。	`atoi`、`atol()`、`atoi()`、`atoll()`、`atoll()` に文字列が渡される場合にこれらを検出する構造ルールを作成した。
INT31-C	高	完了	整数変換によってデータの消失や解釈間違いが発生しないことを保証すること。	C のルールと同じ。代入式の左辺の変数をチェックしない型変換を見つける構造ルールを作成できる。
INT32-C	高	完了	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こと。	C のルールと同じ。対象となる演算が実行され、if 文がないかをテストする構造ルールを作成できる。
INT33-C	低	完了	除算演算およびモジュロ演算がゼロ除算エラーを引き起こさないことを保証すること。	C のルールと同じ。INT32-C に似た構造ルールを作成した。
INT35-C	中	完了	文字入力関数からの戻り値を切り捨てないこと。	`char` 変数に int を返す関数を検出する構造ルールを作成した。
FLP30-C		完了	浮動小数点値の比較時には精度を考慮すること。	C のルールと同じ。
FLP31-C	低	完了	浮動小数点変数をループカウンタとして使わないこと。	ループから抜ける前に、ループ内で変更された浮動小数点をテストする条件付きループを検出する構造ルールを作成した。
FLP32-C	低	完了	数学関数におけるドメインエラーを防止すること。	C のルールと同じ。
ARR00-A		なし	配列のサイズを知るために `sizeof` 演算子を使用するのは避けること。	`sizeof` は Fortify がコードを分析できるようになる前に処理される。
ARR30-C	高	一部	配列のインデックスが確実に適切な範囲内にあるようにすること。	C のルールと同じ。
ARR31-C	高	なし	すべてのソースファイルにわたり一貫性のある配列表記法を	C のルールと同じ。

			使用すること。	
DAN30-C	高	なし	ライフタイムの範囲外のオブジェクトを参照しないこと。	Fortify はスコープの問題は解決できない。
DAN31-C	高	FORTIFY	削除されているオブジェクトにアクセスしないこと。	\[445B3F3C1AB46D8CC28EA535D6436803 :medium :Use After Free :controlflow \] によって捕捉される。
DAN32-C	高	完了	this を削除しないこと。	構造ルールで、this~ という名前の変数を使って呼び出された場合の delete 関数を捕捉する。
DAN33-C	高	なし	無効な反復子を使用しないこと。	これは STL30-C と同じ。
DAN34-C	高	なし	無効なポインタを逆参照しないこと。	Fortify は `new` を検出できない。
ERR31-C	低	なし	デストラクタは例外セーフであること。	Fortify は `throw` を検出できない。
RES30-C	低	なし	1 つの文で複数のリソースを割り当てないこと。	フロントエンドは一時変数を導入し、不適合コードを適合コードに変える予定である（順番は実装で定義される）。このため構造アナライザは、元のコード内のすべての構文パターンは照合できない。導入される一時変数による問題解決の計画はあるが、これは次期リリースの最優先課題ではない。より一般的なルールは「呼び出しパラメータにおいて複数の副次的効果を許可しない」である。これは、より多くの手間を要する。構造解析と構造ルールの背後にある原則は、構造ルールが構文パターンに厳密に一致する必要がある、ということである。このためほとんどの場合、1 つの構造ルールによって意味的に等しいすべてのコードパターンを照合することはできない。1 つのルールで意味的に等しいすべてのコードパターンを照合するという目標を達成するために、我々はより洗練された解析方法を導入する必要があるが、それには多くのオーバーヘッドを伴う可能性がある。
RES31-C	低	FORTIFY	リソースの割り当てはすべて、所有しているオブジェクトに即座にリソースを割り当てる専用の文で	フロントエンドは、次の文 `int i = xxxx;` を次のように変換する。

			実行すること。	`int i;`
				
				`i = xxxx;`
				
				データフローアナライザは、どの実行パスでも初期化されずに使用されている変数をすべてマークする。
				
				Fortify は、\[9C8847DF979C3B2462D6E0C7C30BACB2 :low :Uninitialized Variable :controlflow \] によって NCCE を捕捉する。

RES32-C	高	完了	素のメモリ割り当てや解放ではなく、new や delete を使うこと。	
RES33-C	低	Fortify	オブジェクトと配列の削除は、対応する new と正しく組み合わせること。	\[B0B21546D73D736EF3111D2D80AAA168 :medium :Memory Leak :controlflow \] によって捕捉される。
RES34-C	低	なし	取得と解放の組み合わせが必要なリソースは、オブジェクトでカプセル化すること。	実行方法が不明。制御フロールールを使ってできる可能性はある。
RES35-C	低	なし	リソースを管理するクラス内でコピーコンストラクタ、コピー代入演算子、およびデストラクタを宣言すること。	Fortify はこのルールの違反を検出できない。Fortify ではこれは不可能。
RES36-C	低	なし	コピー代入演算子が、自分自身にコピーされるオブジェクトを壊さないようにすること。	Fortify は、メンバ関数がメンバ変数を削除し、そのあとで変数の内容を使おうとしたことを検出できない。
RES37-C	低	なし	取得と解放の組み合わせが必要なリソースは、オブジェクトのデストラクタで解放する。	Fortify は明示的なデストラクタの有無を検出できない。
RES38-C	低	なし	例外をスローするときにリソースをリークしないこと。	Fortify は `throw` 文を検出できない。
RES39-C	低	完了	`longjmp()` は使用しないこと。	`longjmp()` のすべての呼び出しを検出するセマンティックルールを作成した。
OBJ30-C	高	なし	ポインタ算術演算を多態的に使用しないこと。	実装方法が不明。

OBJ31-C	高	なし	配列を多態的に扱わないこと。	OBJ30-C 参照。

OBJ32-C	高	なし	単一引数のコンストラクタを "explicit" と指定すること。	構造ルールを試みたが、コンストラクタと明示的なコンストラクタの違いを区別できなかった。条件式を無視するため不可能。たとえば次のような代入文の中にあれば、
				`Widget * wt;`
				`w1 = 2;`
				このケースを満たす構造ルールを記述できる。条件式に合うように、新しいラベル「ConditionalExpression」を定義/実装する必要がある。

OBJ33-C	低	一部	多態性オブジェクトをスライスしないこと。	同一と設定されているメンバが属するクラスを拡大するクラスのメンバを検出しない。 考えられる 1 つの方法は、代入文の lhs と rhs の型をチェックすることである。型がプリミティブでない場合、代入によりオブジェクトのスライスが発生する可能性がある。しかし SCA は、初期化における代入と、コード中のそれ以外を区別できないため、必要以上に多くの代入を検出する。
OBJ34-C	中	なし	多態性オブジェクトに対して確実に適切なデストラクタが呼び出されるようにすること。	Fortify は派生クラスと非派生クラスを区別できない。
BSC30-C	低	完了	NULL 終端バイト文字列への const ポインタ取得に `c_str()` メンバを使うこと。	クラス `basic_string` メンバ関数 data のすべての使用を捕捉するセマンティックルールを作成した。
BSC31-C	低	完了	`c_str()` メンバによって返される NULL 終端バイト文字列を変更しないこと。	`c_str()` によって返された文字列が `str*cat()` または `str*cpp()` を使って変更されたことを検出する制御フロールールを作成した。
BSC32-C		完了	`c_str()` によって返されたポインタ値を以降の非 const メンバ関数への呼び出しのあとには使わないこと。	basic_string クラスの非 const メンバ関数が、ポインタ値が `c_str()` によって返されたあとに呼び出されたことを検出する構造ルールを作成した。
BSC34-C	高	なし	要素アクセスの範囲	C ルール ARR30-C は、Fortify

			チェックを行うこと。	の配列アクセスをテストできないようである。
STR30-C	低	なし	文字列リテラルを変更しようとしないこと。	C のルールと同じ。
STR32-C	高	FORTIFY	境界付き文字列をコピーするときには適切な領域を割り当てること。	サンプルコードは Fortify ルール \[577ED976ECB85D475F17575778932434 :high :Buffer Overflow :dataflow \] によって捕捉された。これはサンプルコード全体の結果であると考えられる。
STR35-C	高	なし	固定長配列への読み込み時に入力を制限すること。	制御フロールールの記述を試みた。`cin` または `operator>>` または `>>` を検出できない。
STL30-C	低	なし	有効な反復子を使用すること。	Fortify は異なる型の単項演算子を区別しないようである。 これはフロントエンドによる一時変数の導入の結果である。`d.insert (pos++, data[i]+41)` は `t0 = pos++; d.insert(t0, data[i]+41)` に変換される。次期リリースでは、`t0 = pos++` を照合できる予定だが、まだ内部機能である。
STL31-C	高	なし	有効な反復子範囲を使用すること。	反復子は Fortify がこれを取得する前に処理されるようである。
STL32-C	低	なし	有効な順序付けルールを使用すること。	Fortify はこれを検出できない。
MSC31-C	高	なし	1 つの定義ルールに従うこと。	Fortify はこれを検出できない。

付録 C　COMPASS / ROSE 向け CERT C セキュアコーディングスタンダード拡張チェッカー実装のための補足情報

以下に記述された情報は、解析プロセスのアーティファクトであり、本報告書の本文に提示した解析結果を補足するものである。この情報は完全でもなく、また、決定的なものでもないため、注意深く使用願う。

ルール・推奨事項	深刻度	進捗	説明	備考
DCL30-C	高	一部	適切な記憶域期間でオブジェクトを宣言すること。	Rose は、ローカル変数へのポインタが返されていることを自動的に検出する。またそれ以外にも、静的ポインタへの自動変数の代入など、いくつか特定の例も捕捉できる。
EXP01-A	高	完了	型のサイズを知るためにポインタの sizeof を取得しないこと。	Rose は次のテンプレートコードを検出する。`* T1* x = malloc(sizeof(T2) * y)` `(T1*) malloc(sizeof(T2) * y)` ここで、T1 != T2
EXP08-A	高	なし	ポインタ算術演算は正しく使うこと。	
EXP34-C	高	完了	ポインタを逆参照する前にその有効性を確認すること。	Rose では現在、`malloc()` の結果の受け取り後、ポインタは次に必ず `==` または `!=` 演算（たとえば、*`if (ptr == NULL)...`*）、または bool 型に変換される演算（たとえば *`if (ptr)...`*）の中で使われるようになっている。 Rose は、ptr が単純変数以外（たとえば、構造体メンバ、配列メンバ、逆参照ポインタなど）に代入されるケースには対処できない。
INT13-A	高	なし	右シフト演算が論理シフトまたは算術シフトとして実装されていると想定しないこと。	右シフト演算が実行されたことを検出する構造ルールを作成できる。 Rose で同じことができるが、`>>` 演算自体に問題があるわけではない。`>>` 演算の結果を想定しているかどうかをチェックする方法は判明していない。
INT31-C	高	一部	整数変換によってデータの消失や解釈間	代入式の左辺の変数をチェックしない型変換を見つける構

			違いが発生しないことを保証すること。	造ルールを作成できる。 Rose はすでに、符号変換と整数型について警告を発行する。それらの警告は `limits.h` がインクルードされていると必ず発生するため、完全には信頼できない。
INT32-C	高	なし	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こと。	対象となる演算が実行され、if 文がないかを判定する構造ルールを作成できる。 おそらく Rose で実施できるが、加算がオーバーフローを引き起こさず、チェックできない事例が多数生じる。「妥当な」使用法のチェックを抑えるにはどうしたらよいか？
INT35-C	高	なし	大きなサイズの整数に対して比較や代入を行う場合は事前に整数をアップキャストすること。	AFAICT ROSE は、明示的な型変換と暗黙的な型変換（たとえば、コンパイラによるプロモーションなど）の違いを区別しない。それでもこのルールの順守は可能である。スコープを、外側の演算は代入か比較であり、内側の演算子はそうでない、<exp> <op> (<exp> <op> <exp>) の形式の等式に限定する。内側の演算子に対するオーバーフローチェックは、通常は内側の式の型変換を慎重に扱うことで緩和するべきである。
INT36-C	高	なし	負のビット数、またはオペランドにあるよりも多くのビットをシフトしないこと。	`<<` または `>>` 演算子で使われる変数はすべて、比較式の中で以前に出てきていることを確実にすることができる。これは、静的解析よりも動的解析のほうが常に有効になる 1 つのルールである。
ARR00-A	高	一部	配列のサイズを知るために `sizeof` 演算子を使う場合は慎重に行うこと。	Rose は完全な配列宣言と不完全な配列宣言の違いを区別するが、不完全な配列宣言とポインタ宣言の違いは区別しない。そのため、`sizeof` オペランド型がポインタ（または不完全な配列）であることと、`sizeof` が除算式の除数であることをチェックする。p が除算式にない不完全な配列またはポインタである場合に、`sizeof(p)` を検出しない。これを行う方法があるかどうかは不明。
ARR30-C	高	なし	配列のインデックスが確実に適切な範囲内にあるようにする	配列範囲チェックの一般的な問題に対しては、静的解析よりもはるかに動的解析のほう

			こと。	が適している。
ARR31-C	高	なし	すべてのソースファイルにわたり一貫性のある配列表記法を使用すること。	ROSE では、ポインタ型と不完全な配列型の違いを区別すると仮定すると、これが可能だと考えられる（これに対する ROSE の動作は最近変更された）。1 つのファイル用にこれを作成するのは簡単だが、ファイル間の矛盾を捕捉するためにプロジェクト全体に適用する必要が生じる。
ARR32-C	高	なし	可変長配列へのサイズ引数は、適切な範囲内であること。	配列が動的に割り当てられている場合と値が正しくチェックされていないことを検出する構造ルールを作成した。Fortify は単一関数の適用範囲外を認識できないため、これはサンプルの適合コードを検出する。 配列参照内で使われている変数が比較演算子の中で直前に使われていることを確実にすることはできるが、あまり包括的なものにはならない。また、あるファイルで変数が変更され、そのあと別のファイルの関数に送られて配列を宣言することがあるため、複数のファイルを確認する必要も生じる。
ARR33-C	高	なし	コピーは必ず十分なサイズの記憶領域に対して行われるようにすること。	Rose は NCCE でコードを捕捉する。変数のサイズが妥当であることを確実にするもう 1 つのケースである。これは、おそらく動的に行うほうがよい。この場合、`memcpy` の arg1 に割り当てられるメモリのサイズが、（`memcpy` の arg3 で指定された）データのサイズに合う十分な大きさであることを確実にする必要がある。これは静的にできる可能性があるが、ある値がほかの値よりも大きいことをコンパイル時に判定するインフラストラクチャが必要であり、現時点ではこれが不足している。
ARR34-C	高	一部	式内の配列の型は必ず互換性があるようにすること。	Rose のデフォルトのルールで、すでに互換性のない配列コピーが検出される。このため、gcc も同様である。可変長配列の検出は行われない。おそらく可能であるが、簡単ではない。
STR00-A	高	該当せず	既存の文字列処理コードの修正には TR	

			24731 を使用すること。	
STR01-A	高	該当せず	新しい文字列処理コードの開発には、managed 文字列を使用すること。	
STR31-C	高	一部	文字列の記憶域に文字データと NULL 終端文字用に十分な領域があることを確実にすること。	Rose は、`strcpy` の第 1 引数が固定長配列で宣言されていることを検出する。現在、`strcpy_s` はサポートしていない。（手動の `strcpy` を実行する）最初のサンプルを特定する方法は不明である。
STR32-C	高	なし	すべてのバイト文字列は確実に NULL で終了させること。	複雑だが可能である。文字列から NULL 終端ステータスを削除する可能性のある関数（たとえば、`strcpy`、`strncpy`、`realloc`、`memcpy`）を検索する。このような関数の場合は、文字列長（`strlen( string)`がある）に基づいて、それが if 文の内側にあることを確かめる。または、文字列の次の使用で NULL 終端文字が追加されることを確かめる。それによりすべてのサンプルコードが捕捉されるはずである。
STR33-C	高	完了	ワイド文字の文字列を正しくサイズ指定すること。	このルールは EXP-09-A、および ROSE 自身によってカバーされている。
STR34-C	中	なし	文字をより大きな整数サイズに変換する場合は事前に unsigned 型に変換すること。	`<int> = <char>` を検索し、 `<char>` が先に `<unsigned char>` に型変換されていないか診断できる。
STR35-C	高	なし	制限のないソースから固定長の配列へデータをコピーしないこと。	`gets()`の使用をすべてレポートする。また、`scanf()` 内の %s も捕捉する。`getc()` の例を捕捉する方法は不明。
MEM00-A	高	なし	メモリの割り当てと解放は、同一モジュールの同一抽象レベルで行うこと。	単に `malloc` のみ、または単に `free` のみの fn を捕捉する制御フロールールを簡単に作成できる。しかし、多くのフォールスポジティブが捕捉される。必要なのは、malloc を含む関数のそれぞれを、free を含む関数と対応させることである（これは、C++ と RAII が C に勝る領域の 1 つ）。

MEM01-A	高	なし	動的に割り当てられたメモリへのポインタは、解放後に NULL に設定すること。	解放後のポインタの使用はすべて、代入演算子の左辺にある（たとえば、別の値または NULL に設定されている）ことを確実にするルールを ROSE に追加する。
MEM02-A	低	該当せず	`malloc()` からの戻り値は型変換しないこと。	このルールは変更されている。
MEM04-A	高		0 バイトの割り当ての結果について仮定しないこと。	`malloc arg`、または `realloc arg` の値がゼロでないことを確認する必要があり、そのためには（おそらく）いくつかの変数値のアサーションが必要である（同様の問題については ARR33-C を参照）。
MEM07-A	高	なし	`calloc()` へのサイズ引数が整数オーバーフローを引き起こさないことを確実にする。	`calloc` 引数内で整数オーバーフローの可能性を特定するのは、やや難しい。しかしさらに難しいのは、そのようなオーバーフローを防ぐ、先行するコードを認識することである。
MEM30-C	高	一部	解放したメモリにアクセスしないこと。	Rose は現在、解放されたあとの変数が（代入以外の）関数内で使われていることを検出する。最初のサンプルは捕捉しない。このためより洗練された解析方法が必要であるが、それが重要かはわからない。より包括的な対応には動的解析が必要である。
MEM31-C	高	なし	動的に割り当てられたメモリは必ず一度解放すること。	MEM30-C 用の ROSE コードは重複解放を検出するが、そのルールをコピーして特に重複解放を特定するのは簡単なはずである。
MEM35-C	高	なし	オブジェクトに十分なメモリを割り当てること。	`malloc arg` の内側の乗算がオーバーフローを引き起こさないことを確実にする、巧妙な数学ルールをいくつか作成する必要がある。代わりに、より難しいルールは、オーバーフローを防ぐ、先行するコードを認識することである。
FIO07-A	低	完了	`rewind()` よりも `fseek()` を使用すること。	
FIO12-A	低	完了	`setbuf()` よりも `setvbuf()` を使用すること。	

FIO30-C	高	なし	書式文字列からユーザ入力を除外すること。	書式文字列として使われている変数の内容の出処を突き止めるのは、おそらく非常に困難である。おそらく、有効な方法は、`printf`（およびその他の書式関数）内の変数書式文字列の使用を検出することである。*ただし*、その変数の以前の使用を定数文字列に初期化する場合を除く。これは i18n の妨げになるが、i18n は以前から安全でない書式文字列に対して非常に脆弱である。
FIO34-C	高	なし	文字入出力関数の戻り値の取得には int を使用すること。	サンプルに基づくと、ここでの適切なルールは、EOF から符号なし整数への暗黙的な型変換をすべて検出することである。あるいは、型変換、EOF と符号なし整数の比較、または `char` の無視をすべて検出する必要がある。
FIO35-C	高	なし	end-of-file およびファイルエラーの検出には `feof()` および `ferror()` を使用すること。	ここでは、EOF と `getchar()`（または同様の関数）の結果との比較をすべて検出する必要がある。または、`getchar()`（または同様の関数）の結果が最後に代入された変数との比較をすべて検出する。FIO34-C と非常によく似ているがまったく同じルールではない。
FIO36-C	高	なし	`fgets()`の使用時に改行文字が読み取られると想定しないこと。	`fgets()` 呼び出しの最後の文字が改行であると、コードが想定しているかを知る方法があるかはわからない。
FIO37-C	高	なし	文字データが読み取られたと想定しないこと。	FIO36-C と同様に、Rose ルールでは、プログラマが行った暗黙的な想定が何かを知る必要がある。
FIO43-C	高	なし	制限のないソースから固定長の配列へデータをコピーしないこと。	STR35-C の重複のように見える。
TMP00-A	高	なし	共有ディレクトリに一時ファイルを作成しないこと。	簡単な検出事例が多数ある。`mktemp()` および関連する関数を見つけられる。"/tmp" や "c:/TMP"、または同様のパターンで `const` 文字列も見つけられる。しかしどれがこのルールに対する適合コードサ

				ンプルになるか？これを行う正しい方法は現時点ではないと思われる。
TMP30-C	高	なし	一時ファイルは一意かつ予測できないファイル名で作成すること。	ルールは簡単に順守できる必要があるが、適合する解決法については自信がない。このルールは また、TMP00-A と大部分が重なる。
TMP32-C	高	完了	一時ファイルは排他的アクセス権で開くこと。	Rose は現在、`tmpfile()` のすべてのインスタンスを検出する。`fopen()` のあとの `tmpnam()`、`fopen_s()` のあとの `tmpnam_s()`、`open()` のあとの `mktemp()` の使用もすべて検出する（同じ変数を使うと仮定する）。
ENV01-A	高	なし	環境変数のサイズについて仮定しないこと。	ここでの解決法は、`getenv()` の結果を受け取る文字列（`char[]` または `char*`）のすべての使用をチェックすることである。2 番目の文字列が、`strlen(string1)` がかかわる `malloc` を使って割り当てられていない限り、この文字列が別の文字列へ標準コピーされるのを許可しないこと。
ENV04-A	高	完了	コマンドインタプリタが必要ない場合は system() を呼び出さないこと。	Rose は `system()` の呼び出しをすべて検出する。ユーザがコマンドインタプリタを必要としていることを認識できない。
MSC30-C	低	完了	rand 関数を使用しないこと。	
POS33-C	低	完了	`vfork()` を使用しないこと。	
POS34-C	高	完了	引数に自動変数を使って `putenv()` を呼び出さないこと。	Rose は配列 arg を使って `putenv()` を検出する。（`ptr arg` は検出されない）。静的配列を間違って検出する。不適切な ptr の使用法を間違って見逃す（ただし MEM00-a はこれらを捕捉するはずである）。

付録 D　COMPASS / ROSE 向け CERT C++ セキュアコーディングスタンダード拡張チェッカー実装のための補足情報

以下に記述された情報は、解析プロセスのアーティファクトであり、本報告書の本文に提示した解析結果を補足するものである。この情報は完全でもなく、また、決定的なものでもないため、注意深く使用願う。

ルール・推奨事項	深刻度	進捗	説明	備考
DCL30-C	低	一部	実装用に予約された名前を使わないこと。	プラットフォームごとのルールの設定が困難であるため現在無効。設定なしの場合、多数のフォールスポジティブ。
DCL32-C		一部	外部リンクを使ったオブジェクトのランタイム静的初期化を避けること。	定義されていない外部宣言に対するフォールスポジティブのため、現在無効。
EXP00-A		一部	C のスタイルの型変換を使用しないこと。	フォールスポジティブのため現在無効。
EXP02-A		完了	論理 AND 演算子および論理 OR 演算子をオーバーロードしないこと。	
EXP03-A		完了	& 演算子をオーバーロードしないこと。	
EXP04-A		完了	コンマ演算子をオーバーロードしないこと。	
EXP08-A		完了	すべての列挙値をテストするのでない限り、switch 文には default 句を入れること。	
EXP09-A		完了	関係演算子と等価演算子を非結合のように扱うこと。	
EXP10-A		一部	++ および -- のプレフィックス形式が望ましい。	バグのため無効。
EXP36-C	中	一部	不完全クラスへのポインタを型変換および削除しないこと。	バグのため無効。
EXP38-C		完了	コンストラクタおよびデストラクタ内で独自の仮想関数を呼び出さないこと。	

EXP39-C		一部	クラスオブジェクトのビット単位コピーを行わないこと。	バグのため無効。
ERR01-A		完了	例外に特殊用途の型を使うこと。	
ERR02-A		完了	匿名の一時オブジェクトをスローし、参照で捕捉すること。	
RES35-C	低	完了	リソースを管理するクラス内でコピーコンストラクタ、コピー代入演算子、およびデストラクタを宣言すること。	
OBJ00-A		完了	データメンバをプライベートとして宣言すること。	
OBJ01-A		完了	変換演算子の定義に注意すること。	
OBJ02-A		完了	継承された非仮想メンバ関数を隠蔽しないこと。	
OBJ03-A		完了	仮想関数のオーバーロードを避けることが望ましい。	
OBJ04-A		完了	仮想関数にはデフォルトの引数イニシャライザを指定しないことが望ましい。	
OBJ32-C	高	完了	単一引数のコンストラクタを "explicit" と指定すること。	

付録 E　CERT セキュアコーディングルール・推奨事項に設定された優先度とレベルについての補足

各ルールと推奨事項には優先度が設定されている。優先度の設定には、故障モデル(failure model)、効果(effect)、致命度解析(criticality analysis)(FMECA)［IEC 60812］に基づくメトリックを用いている。各ルールには、次の 3 項目について 1 から 3 のスケールで 3 つの評価値を割り当てている。

- 脅威度：ルールが無視された場合の結果がどれくらい深刻か

  1. 低 (DoS 攻撃、危険な終了)
  2. 中 (データの完全性違反, 意図しない情報漏えい)
  3. 高 (任意のコード実行)

- 可能性：ルールを無視することで作りこまれた欠陥が攻撃可能な脆弱性につながる可能性

  1. 低
  2. 中
  3. 高

- 修正コスト：ルールに適合するために要するコスト

  1. 高 (手作業で発見・修正する)
  2. 中 (自動で発見し、手作業で修正する)
  3. 低 (自動で発見・修正する)

これら 3 つの評価値は次にそれぞれ掛け合わされる。掛け算の結果はルールの適用を優先度づけするための指標となる。値は 1 から 27 の範囲をとる。優先度が 1 から 4（P1-P4）の範囲のルールと推奨事項はレベル 3（L3）、優先度が 6 から 9（P6-P9）はレベル 2（L2）、優先度が 12 から 27（P12-P27）はレベル 1（L1）となる。次の図で示すように、すべてのルールと推奨事項にレベルを設定することで、レベル 1、レベル 2、もしくはスタンダードへ完全に適合(レベル 3)を主張することができる。

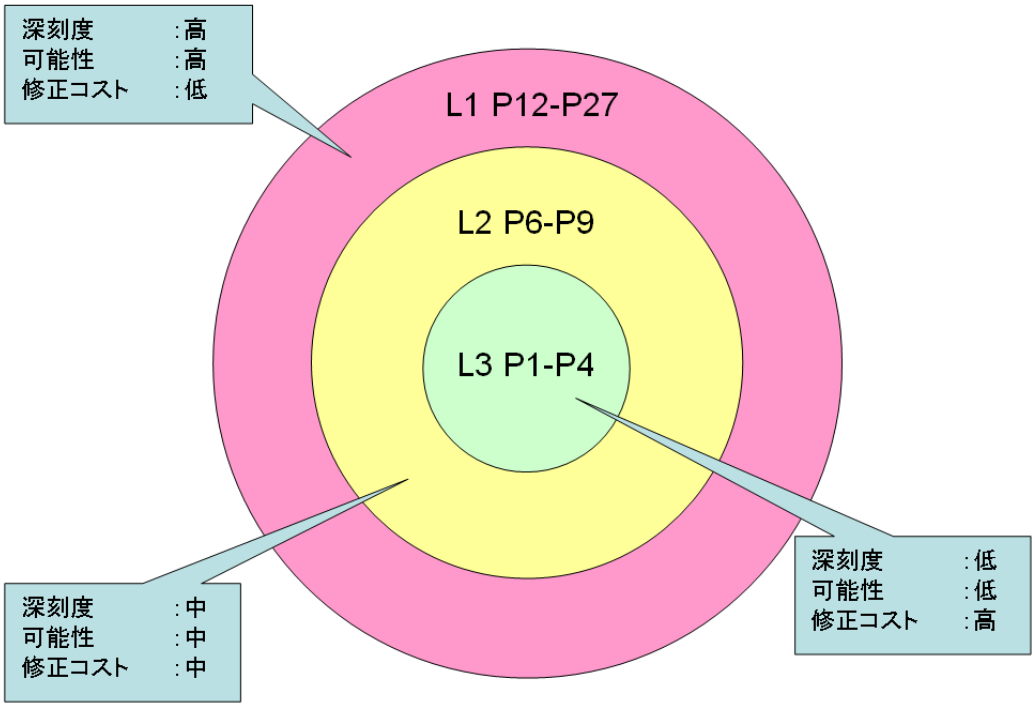

図 3：CERT セキュアコーディングルール・推奨事項に設定された優先度とレベル

参考文献

URL は本書発行日時点で有効である。

[Almossawi 06] Almossawi, A.; Lim, K; & Sinha, T. “Analysis Tool Evaluation:Coverity Prevent.” Pittsburgh, PA:Carnegie Mellon University, 2006. http://www.cs.cmu.edu/~aldrich/courses/654/tools/cure-coverity-06.pdf.

[CERT 07a] CERT. “CERT C Programming Language Secure Coding Standard.” Pittsburgh, PA:Software Engineering Institute, CERT, 2008. https://www.securecoding.cert.org/confluence/x/HQE.

[CERT 07b] CERT. “CERT C++ Programming Language Secure Coding Standard.” Pittsburgh, PA:Software Engineering Institute, CERT, 2008. https://www.securecoding.cert.org/confluence/x/fQI.

[CERT 07c] CERT. “CERT Statistics.” Pittsburgh, PA:Software Engineering Institute, CERT, 2008. http://www.cert.org/stats/cert_stats.html.

[Chess 02] Chess, B. “Improving Computer Security using Extended Static Checking,” ***Proceedings*** *of the* ***2002 IEEE Symposium*** *on* ***Security and Privacy***.Los Alamitos, CA :IEEE CS Press, 2002.

[ISO/IEC 9899-1999] ISO/IEC.*Programming Languages − C, Second Edition* (ISO/IEC 9899-1999).Geneva, Switzerland:International Organization for Standardization, 1999.

[ISO/IEC 14882-2003] ISO/IEC.*Programming Languages − C++, Second Edition* (ISO/IEC 14882-2003).Geneva, Switzerland:International Organization for Standardization, 2003.

[ISO/IEC TR 24731-1-2007] ISO/IEC TR 24731. *Extensions to the C Library − Part I:Bounds-checking interfaces*.Geneva,

Switzerland:International Organization for Standardization, April 2006.

[Larochelle 01] Larochelle, D. & Evans, D. "Statically Detecting Likely Buffer Overflow Vulnerabilities," *Proceedings of the 10th Usenix Security Symposium* (USENIX' 01).Berkeley, CA:Usenix Association, 2001.

[Seacord 05] Seacord, R. *Secure Coding in C and C++*.New York, NY:Addison-Wesley, 2005.

[US-CERT 08] US-CERT. "US-CERT Technical Cyber Security Alerts." Washington, DC, 2008. http://www.us-cert.gov/cas/techalerts/index.html.

[Wallnau 01] Wallnau, K.; Hissam, S.; & Seacord, R. *Building Systems from Commercial Components*.New York, NY:Addison-Wesley, 2001.