Vol.2013-DPS-154 No.12
Vol.2013-CSEC-60 No.12
2013/3/14

汚染解析実施後のサニタイザ適切配置支援に関する考察

小黒 博昭[1]　橋本 卓哉[1]

概要：Web アプリケーションの脆弱性をソースコードレベルで検査する手法として，セキュリティに特化した静的解析ツールが広く知られており，その商用ソフトウェアも市場に広く流通している．これらの静的解析ツールは，通常，ソースコード上の外部入力点（Source; ソース）から，画面出力点やデータベースへのクエリーが送信される点などのセキュリティ上考慮されるべき出力点（Sink; シンク）に至るデータフローを解析し，そのすべてのパスにおいて，出力点に応じたエスケープ処理（Sanitizer; サニタイザ）が適用されているかという観点で検査される．ツールは，ソースからシンクへのパスの一部またはすべてを利用者へ示し，サニタイザが適用されているかの確認を利用者へ促したり，事前にツールの診断定義ファイルに登録されたサニタイザ関数名を持つ関数の適用がパス上に存在するかを検査するなどして，脆弱性の可能性がある箇所を出力する．このような解析は汚染解析（Taint Analysis）と呼ばれる．しかしながら，従来の静的解析ツールは，脆弱性の可能性がある箇所の出力までしか行わないものがほとんどであり，そのため，汚染解析実施後に脆弱性が発見された場合，サニタイザを適切に配置するように修正する作業は人間が注意深く行う必要があり，これにかかるコストが大きいという問題がある．本稿では，静的解析ツールが診断対象のソースコードに即してサニタイザの適切な配置箇所を利用者へ提案するための手法を提案する．

1. はじめに

Web アプリケーション脆弱性に関する調査研究およびその情報公開を通してソフトウェアのセキュリティ確保を推進する非営利団体 OWASP が 2010 年に公開した脆弱性のトップ 10（OWASP Top 10 [5]）では，1 位がインジェクション，2 位がクロスサイトスクリプティング (Cross-Site Scripting; XSS) となっている．1 位のインジェクションの種類には，SQL インジェクション，OS コマンドインジェクション，等がある．2 位の XSS は HTML タグインジェクションとも見なすことができることから，全般的に，インジェクション系脆弱性への対策がより重要な状況となっている．

Web アプリケーションの脆弱性をソースコードレベルで検査する手法として，セキュリティに特化した静的解析ツールが広く知られており，その商用ソフトウェアも市場に広く流通している．これらの静的解析ツールは，通常，ソースコード上の外部入力点（Source; ソース）から，画面出力点やデータベースへのクエリーが送信される点などのセキュリティ上考慮されるべき出力点（Sink; シンク）に至るデータフローを解析し，そのすべてのパスにおいて，出力点に応じたエスケープ処理（Sanitizer; サニタイザ）が適用されているかという観点で検査される．ツールは，ソースからシンクへのパスの一部またはすべてを利用者へ示し，サニタイザが適用されているかの確認を利用者へ促したり，事前にツールの診断定義ファイルに登録されたサニタイザ関数名を持つ関数の適用がパス上に存在するかを検査するなどして，脆弱性の可能性がある箇所を出力する．このような解析は汚染解析（Taint Analysis）と呼ばれる．しかしながら，従来の静的解析ツールは，脆弱性の可能性がある箇所の出力までしか行わないものがほとんどであり，そのため，汚染解析実施後に脆弱性が発見された場合，サニタイザを適切に配置するように修正する作業は人間が注意深く行う必要があり，これにかかるコストが大きいという問題がある．

本稿では，静的解析ツールが診断対象のソースコードに即してサニタイザの適切な配置箇所を利用者へ提案するための手法を提案する．

本稿の以降の構成は以下の通りである．2 節で，Livshits ら [4] が形式化したサニタイザ配置問題を導入し，彼らが提案したノードベースアプローチを述べる．3 節で，ソフトウェア開発の現実的観点から，同ノードベースアプローチに対する課題を述べ，4 節でその課題を解決する一方法

[1] 株式会社 NTT データ 基盤システム事業本部 セキュリティビジネス推進室
NTT DATA Corporation, Toyosu Center Bldg. Annex, 3-3-9, Toyosu, Koto-ku, Tokyo 135–8671, Japan

を提案し，5 節でその考察を述べる．6 節で関連研究を述べ，7 節でまとめと今後の課題を述べる．

2. サニタイザ適切配置問題

本節では，文献 [4] で形式化されたサニタイザ適切配置問題に関する各種定義を示す．

2.1 ソース，シンク，サニタイザ

本節では，本稿で用いる用語であるソース，シンク，およびサニタイザの定義および例をそれぞれ示す．これらの定義は文献 [4] で特別に導入されたものではなく，汚染解析の研究領域において，一般的に認識されている概念である．

定義 1 (ソース) ソースとは，外部から入力されるデータをプログラム [*1] に取り込む処理，またはその処理が記述されたプログラム上の位置である．

例 1 Java[*2] 言語における J2EE では，HttpServletRequest オブジェクトの getParameter メソッドや getAttribute メソッドは代表的なソースである．

定義 2 (シンク) シンクとは，もしそこで危険な文字列が出力された場合に意図しないセキュリティ侵害が発生する可能性がある処理，またはその処理が記述されたプログラム上の位置である．

例 2 Java 言語において，クロスサイトスクリプティング脆弱性のシンクは，HttpServletResponse オブジェクトの getWriter メソッドや getOutputStream メソッドが代表的である．SQL インジェクション脆弱性のシンクは，java.sql.Statement オブジェクトの execute メソッド，executeQuery メソッド，および executeUpdate メソッドが代表的である．

定義 3 (サニタイザ) サニタイザとは，シンクに渡される可能性があるデータが，シンクで出力された後に意図しないセキュリティ侵害を発生させないよう，シンクの出力先で特殊な意味を持つ文字列を，それをエスケープした文字列に変換する関数（またはメソッド）またはそれが記述されたプログラム上の位置である．

例 3 **表 1** に示される HTML エスケープの例に従い，対象文字列をエスケープし，非対称文字列をそのままとする文字列変換処理関数はサニタイザの一種である．また，**表 2** に示される JavaScript エスケープの例に従い，対象文字列をエスケープし，非対称文字列をそのままとする文字列変換処理関数はサニタイザの一種である．

2.2 データフローグラフ

本節では，文献 [4] に基づくデータフローグラフを示す．

[*1] 本稿では，コンパイル前のプログラムコードを表す「ソースコード」という表現はここで定義する「ソース」と混同しやすいため用いず，「プログラム」という表現を用いる．

[*2] Java および Java 関連の商標は，Oracle やその関連会社の米国およびその他の国における商標または登録商標です．

表 1 HTML エスケープの例
Table 1 An Example of HTML Escapes.

対象文字	エスケープ後の文字列
&	&amp;
>	&gt;
<	&lt;
"	&quot;
’	&#39;

表 2 JavaScript エスケープの例
Table 2 An Example of JavaScript Escapes.

対象文字	エスケープ後の文字列
’	\’
"	\"
\	\\
0x0d	\r
0x0a	\n
/	\/
>	\x3e
<	\x3c

表 3 図 1 におけるサニタイザポリシーの例
Table 3 An Example of Sanitizer Policies in Fig. 1.

		シンク		
		●	■	⊘
ソース	○	S_1	S_2	$\bot$
	□	S_1	$\bot$	$\bot$
	◇	$\bot$	S_2	$\bot$
	⊘	$\bot$	$\bot$	$\bot$

データフローグラフの例を **図 1** に示す．図 1 に示すデータフローグラフは，本稿全体を通しての例として用いる．データフローグラフ $G = \langle N, E \rangle$ は有向グラフであり，ノードの集合 N およびエッジの集合 E からなる．ノードは，データを参照し新たなデータを定義する処理またはその処理が記述されたプログラム上の位置を表し，エッジはデータが処理される流れを表す．本稿では，ソースを表すノードの集合をグラフの上部に記述し，シンクを表すノードの集合をグラフの下部に記述する．

2.3 サニタイザポリシー

本節では，文献 [4] に基づくサニタイザポリシーを示す．

サニタイザポリシーの例を **表 3** に示す．サニタイザポリシーは，ソースのタイプ（○，□，◇）が最左列に記され，シンクのタイプ（●，■）が最上行に記される．⊘ は，セキュリティと無関係の特殊なソースタイプおよびシンクタイプを表す．図 1 に示すように，ソースタイプの例としては，ユーザ入力，ホスト内データ，および外部サイトデータ等が挙げられる．シンクタイプの例としては，HTML への出力，および HTML に含まれる JavaScript 領域（<script> から </script> までの領域）への出力等

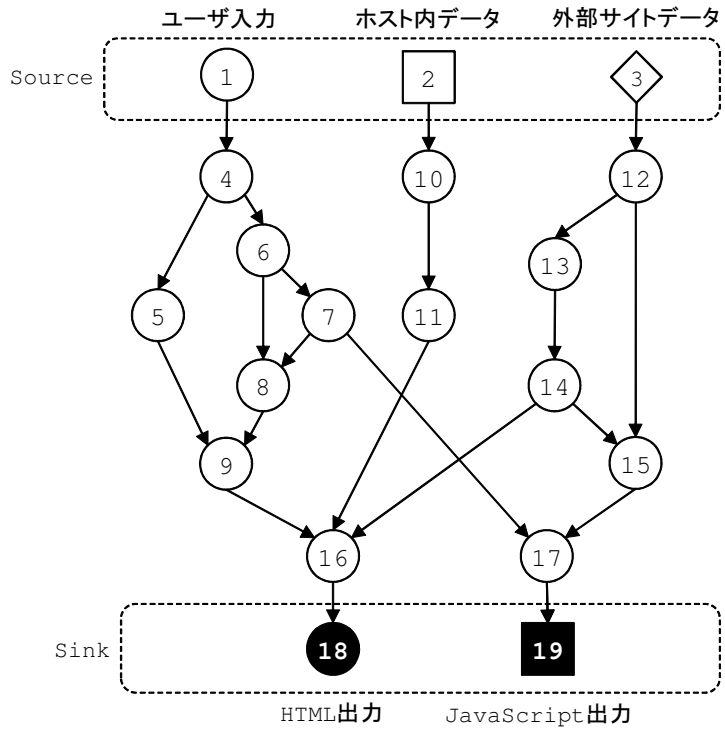

図 1 データフローグラフの例
Fig. 1 An Example of Dataflow Graphs.

が挙げられる．

サニタイザポリシーを表すメタ変数を $\mathcal{P}$, ソースを表すメタ変数を I, シンクを表すメタ変数を O として，$\mathcal{P}(I,O)$ でサニタイザポリシー $\mathcal{P}$ 中の I と O の交点に位置するサニタイザを表す．例えば，表 3 において，$\mathcal{P}(\bigcirc, \bullet) = S_1$ である．これは，○のタイプのソースから●のタイプのシンクに至る経路では，サニタイザ S_1 が適用されるべきであり，その他のサニタイザは適用されてはならないことを規定する．また，$\mathcal{P}(I,O) = \bot$ は，ソースタイプ I からシンクタイプ O に至る経路では，いかなるサニタイザも適用されてはならないことを規定する．I または O のどちらかが $\oslash$ の場合は常に $\mathcal{P}(I,O) = \bot$ であるが，これは，定数データはいかなるサニタイザも適用されてはならないことに相当する．

ノードはソースとシンクに同時にはなり得ないと想定する．

ノード n がソースまたはシンクであるとき，$\tau(n)$ でそのソースタイプまたはシンクタイプを表すとする．例えば，図 1 において，番号 i のノードを n_i と表すと，$\tau(n_1) =$ ○，$\tau(n_{18}) =$ ●である．$\tau(n_4)$ は，n_4 がソースでもシンクでもないため，未定義である．

2.4 サニタイザの妥当性

本節では，文献 [4] に基づくサニタイザの妥当性の定義を示す．

定義 4 (サニタイザの妥当性) $G = \langle N, E\rangle$ をデータフローグラフとする．すべてのソースノード $s \in N$ およびすべてのシンクノード $t \in N$ について，以下の 2 項目が満たされるとき，G に対するサニタイザの適用は，**サニタイザポリシー $\mathcal{P}$ に対して妥当である**と言う．

- $\mathcal{P}(\tau(s), \tau(t)) = S$ ならば，s から t を流れるすべての値にはサニタイザ S がちょうど一回適用され，かつその他のサニタイザは適用されない．
- $\mathcal{P}(\tau(s), \tau(t)) = \bot$ ならば，s から t を流れるすべての値はいかなるサニタイザの適用を受けない．

一般に，サニタイザはすべてのパスにおいて，それが適用されるならば高々 1 回でなければならない．なぜなら，サニタイザは文字列から文字列への関数 f であり，ある文字列 x に対し，あるサニタイザ f_1 を適用した文字列 $f_1(x)$ に任意のサニタイザ f_2 (f_2 は f_1 と同じか異なるかどちらでもよい) を適用した文字列 $f_2(f_1(x))$ は $f_1(x)$ に等しくなるとは限らないからである．例えば，クロスサイトスクリプティングの対策の一つである HTML エスケープ（表 1）では，文字 `&` をエスケープした文字列 `&` にもう一度同じエスケープを適用した場合，`&ampamp;` となり，本来意図した文字 `&` の意味が失われ，文字列 `&` の意味に変化してしまう．

2.5 ノードベースアプローチによるサニタイザ配置

本節では，文献 [4] で導入されたノードベースアプロー

表 4　図 1 における S_i-可能ノード，S_i-排他的ノード，および S_i-最遅排他的ノード
Table 4　S_i-Possible, S_i-Exclusive, and S_i-Latest-Exclusive Nodes in Fig. 1

	可能 (possible)	排他的 (exclusive)	最遅排他的 (latest-exclusive)
S_1	1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 16	2, 5, 8, 9, 10, 11	9, 11
S_2	1, 3, 4, 6, 7, 12, 13, 14, 15, 17	15, 17	17
$\perp$	3, 12, 13, 14, 16	—	—

チによるサニタイザ配置の方法を示す．

定義 5 (S_i-可能)　あるソースノード $s \in N$ およびシンクノード $t \in N$ が存在し，s から t へのパス上にノード $n \in N$ が存在し，かつ $\mathcal{P}(\tau(s), \tau(t)) = S_i\,(i = 1, 2, \ldots)$ となるとき，n は **S_i-可能** (S_i-*possible*) であると言う．

定義 6 (S_i-排他的)　ノード $n \in N$ が S_i-可能であり，かつすべてのソースノード $s \in N$ およびすべてのシンクノード $t \in N$ について，n が s から t へのパス上にあるならば $\mathcal{P}(\tau(s), \tau(t)) = S_i$ となるとき，n は **S_i-排他的** (S_i-*exclusive*) であると言う．

直観的に言うと，ノード n が S_i-排他的であるとは，n が S_i-可能であり，かつすべての $j \neq i$ に対して n が S_j-可能ではないことである．

例 4　図 1 に示されるデータフローグラフにおけるサニタイザ S_1, S_2 および $\perp$ に対する可能ノードおよび排他的ノードは，**表 4** の通りである．

ノード n_4 は S_1-可能であると同時に S_2-可能でもある．なぜなら，n_4 は n_1 から n_{18} に至るパスおよび n_1 から n_{19} に至るパスの両方の上に存在し，$\tau(n_1) = \bigcirc$, $\tau(n_{18}) = \bullet$, $\tau(n_{19}) = \blacksquare$, $\mathcal{P}(\tau(n_1), \tau(n_{18})) = S_1$, $\mathcal{P}(\tau(n_1), \tau(n_{19})) = S_2$ であるからである．

ノード n_{17} は，n_1 から n_{19} に至るパスおよび n_3 から n_{19} に至るパスの両方の上に存在するが，S_2-排他的である．なぜなら，$\tau(n_1) = \bigcirc$, $\tau(n_3) = \diamondsuit$, $\tau(n_{19}) = \blacksquare$, $\mathcal{P}(\tau(n_1), \tau(n_{19})) = \mathcal{P}(\tau(n_3), \tau(n_{19})) = S_2$ であるからである．

データフローグラフが比較的疎な状態のときは，S_i-排他的ノードは多く存在するが，比較的密な状態のときは，S_i-排他的ノードは少数となる傾向があり，ある S_i に対してその S_i-排他的ノードが存在しない場合が発生する可能性がある．比較的疎な状態のデータフローグラフにおいて，S_i-排他的ノードはサニタイザの配置先としての良い候補となり得る．しかし，ある一つのパスにおいて，S_i-排他的ノードが複数存在する可能性がある．その場合，それらの中から一つの S_i-排他的ノードを選択してサニタイザの配置先とする必要がある．選択基準としては，可能な限り遅い時点でサニタイザが適用されることが好ましい．なぜなら，一般に，サニタイザが適用されたデータは，適用される前のデータよりデータ長が増加するためである．従って，以下の S_i-最遅排他的の概念が定義される．

定義 7 (S_i-最遅排他的)　ノード $n \in N$ が S_i-排他的であり，かつすべてのソースノード $s \in N$ およびすべてのシンクノード $t \in N$ について，n が s から t へのパス上にあるならば n がそのパス上の最後の S_i-排他的ノードとなるとき，n は **S_i-最遅排他的** (S_i-*latest-exclusive*) であると言う．

例 5　図 1 に示されるデータフローグラフにおけるサニタイザ S_1, S_2 および $\perp$ に対する最遅排他的ノードは，表 4 の通りである．

ノード n_9 および n_{11} は S_1-最遅排他的ノードである．ノード n_{17} は S_2-最遅排他的ノードである．ノード n_{15} は S_2-最遅排他的ノードではない．なぜなら，n_{15} より後方のパス上に，S_2-排他的ノード n_{17} が存在するからである．

定義 7 より明らかに，$\mathcal{P}(\tau(s), \tau(t)) = S_i$ となるすべてのソースノード s およびすべてのシンクノード t において，s から t に至るパス上に S_i-最遅排他的ノードが存在するならば，それは高々一つである．しかしながら，同パス上に S_i-最遅排他的ノードが存在しない場合がある．その場合，同パス上のどのノード上にサニタイザを配置しても，他のパスのサニタイザを干渉することとなるため，適用することができない．従って，S_i-最遅排他的ノードにのみサニタイザを配置する方法では，妥当なサニタイザ配置を与えることができない場合がある．

文献 [4] では，まずノードベースアプローチでサニタイザ配置を試み，サニタイザポリシーに対して妥当となるような配置を与えることができない場合は，ランタイム解析（動的解析の一種）を併用したエッジベースアプローチにより，与えられたサニタイザポリシーに対して妥当となるサニタイザ配置が常に得られると主張されている．

文献 [4] の方法を 図 1 の例に適用した場合，この例ではデータフローグラフがそれほど密ではなく，かつサニタイザポリシーもそれほど複雑ではないため，ノードベースアプローチのみにより解が得られる．すなわち，**図 2** のように，すべての S_i-最遅排他的ノードにサニタイザを配置することにより妥当なサニタイザ配置が得られる．ノード n_3 から n_{18} に至るパス上にはサニタイザが存在しないが，これは正しい．なぜなら，$\tau(n_3) = \diamondsuit$, $\tau(n_{18}) = \bullet$, $\mathcal{P}(\tau(n_3), \tau(n_{18})) = \perp$ であるからである．

3. 課題

本節では，文献 [4] のノードベースアプローチに対し，実際のソフトウェア開発の観点から見た場合の課題を示す．

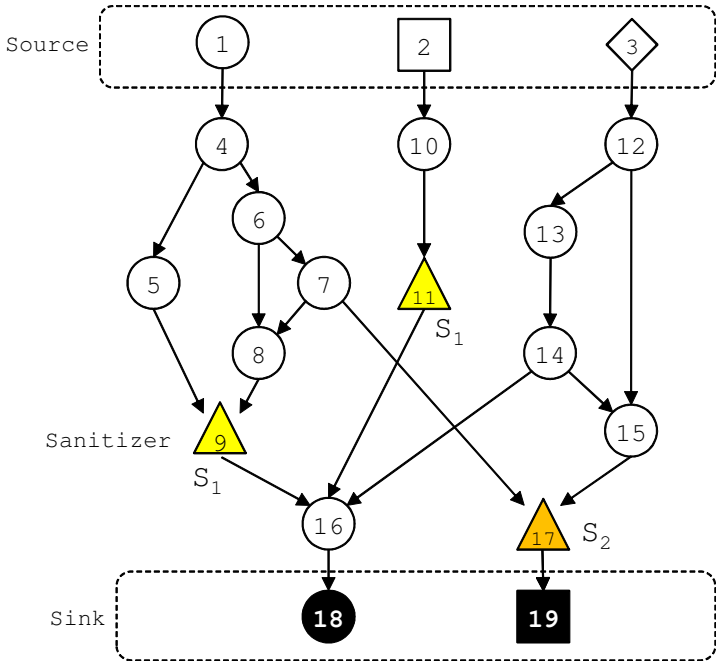

図 2 妥当なサニタイザ配置の例
Fig. 2 An Example of Valid Sanitizer Placements.

一般に，あるサニタイザポリシーに対して妥当となるサニタイザ配置は複数存在する．文献 [4] の方法では，S_i-排他的ノードがサニタイザの配置点の候補とされており，その中でも，S_i-最遅排他的ノードが優先度の高い配置先の候補とされている．

サニタイザを S_i-最遅排他的ノードへ配置する方針は，プログラム記述量の減少およびロジックの単純化を期待でき，プログラムの維持管理の観点から貢献し得るアプローチであると考えられる．さらに，メモリの使用効率や実行速度の観点からも理想的であると考えられる．スクラッチからソフトウェアを開発する場面では，サニタイザを S_i-最遅排他的ノードへ配置する方針は有効であろう．

しかしながら，ソフトウェア開発に関わる様々な費用対効果，環境要因，価値基準を勘案した場合，S_i-最遅排他的ノードへ配置されていなくても，その他の複数の S_i-排他的ノードにサニタイザが配置されることで論理的に S_i-最遅排他的ノードへの配置と同じ効果を期待できるとき，あえて複数の S_i-排他的ノードに配置する方針が採択される場面もあると考えられる．

例えば，スクラッチ開発ではなく，既に開発されたソフトウェアに対する改修の場合を考える．今，対象のプログラムに対する脆弱性検査の結果，妥当ではないサニタイザ配置が**図 3** に示されるように発見されたとする．図 3 では，S_1-最遅排他的ノードの一つである n_9 の位置にサニタイザが配置されておらず，S_1-排他的ノードの一つである n_5 にサニタイザが配置されている．これにより，サニタイザの適用を受けないパス $n_1 \to n_4 \to n_6(\to n_7) \to n_8 \to n_9 \to n_{16} \to n_{18}$ が存在する．このようなパスは，静的解析ツールを用いることにより発見されることが多い．しかしながら，一般的な静的解析ツールでは，誤検出 (False Positive) を削減するため，事前にツールの診断定義ファイルに登録されたサニタイザ関数名を持つ関数が適用されているパスを，検出結果に含めないようにすることができ，そのような使用が推奨されている．もしそのように使用していた場合，前記パスしか検出されず，n_5 を通るパスは検出に含まれないため，n_5 の位置に S_1 が既に配置されていることに気付かない可能性がある．その結果，開発者は誤って，**図 4** のように，n_9 の位置に S_1 を新規に配置するような改修を実施してしまう可能性がある．

実際に，n_9 は S_1-最遅排他的ノードであり，改修に当たる開発者にとっては，n_9 の位置が S_1 の配置先の有力候補に見えてしまう可能性がある．

4. 提案法

本節では，S_i-最遅排他的ノードではない複数の S_i-排他的ノードにサニタイザを配置することにより，S_i-最遅排他的ノードにサニタイザを配置した場合と同等の効果を得るためのサニタイザ配置法を提案する．

はじめに，提案法では以下の前提を置く．

前提 1 データフローグラフは比較的疎であり，かつサニタイザポリシーも比較的複雑ではなく，文献 [4] のノードベースアプローチにより，妥当なサニタイザ配置の解が得られる状態である．

前提 2 ある S_i-最遅排他的ノードにサニタイザが配置されておらず，その上流の S_i-排他的ノードにサニタイザが配置されている．

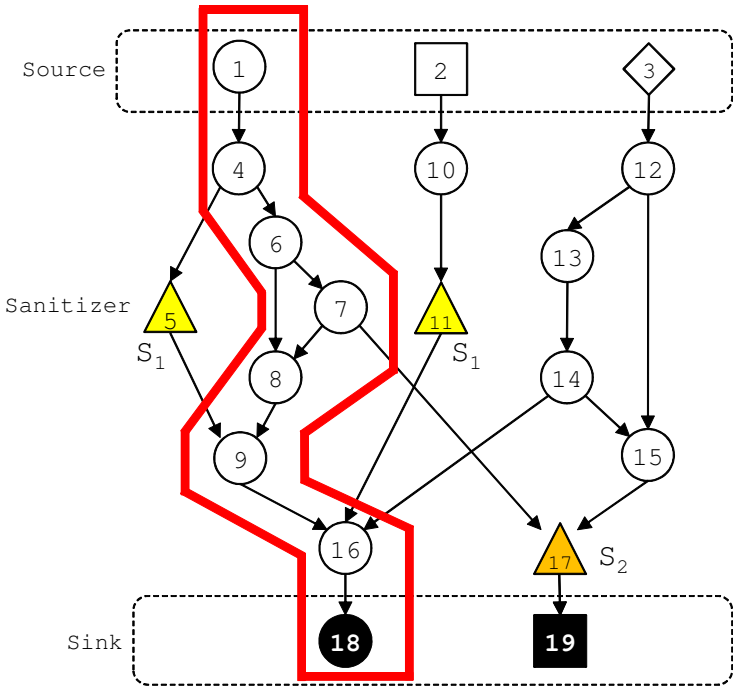

図 3 妥当ではないサニタイザ配置の例
Fig. 3 An Example of Invalid Sanitizer Placements.

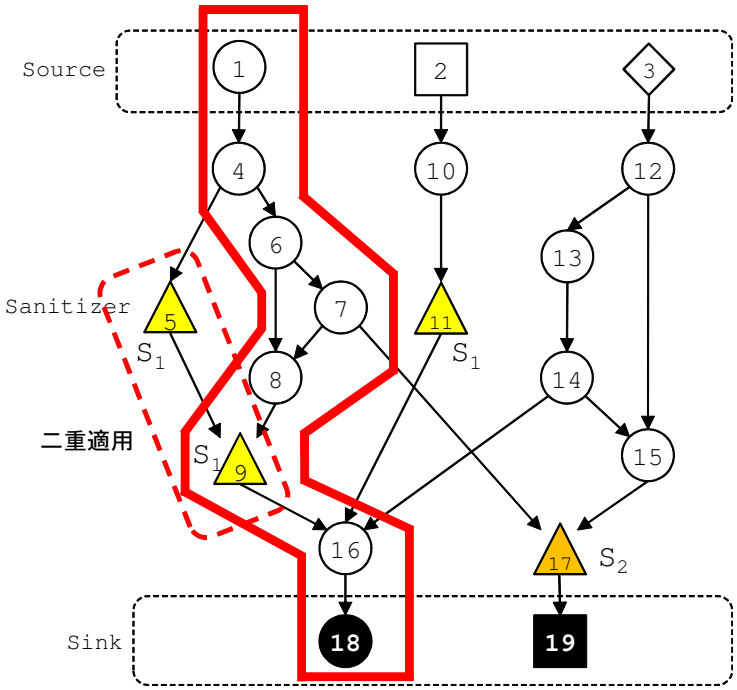

図 4 サニタイザの二重適用が発生する配置例
Fig. 4 An Example of Double Sanitizer Placements.

前提 3 S_i-最遅排他的ノードを含むあるパスに，サニタイザを含まないパスが存在する．

図 3 の状態は上記前提 1, 2, 3 を満足する典型例である．すなわち，ノードベースアプローチにより妥当なサニタイザが得られる状態であり，S_1-最遅排他的ノード (n_9) にサニタイザが配置されておらず，n_9 より上流の S_1-排他的ノード (n_5) にサニタイザが配置されている．n_9 を含むパスにはサニタイザを含まないパスが存在する．

以下に，図 3 の状態から提案法を適用するステップを**図 5** に示しながら，提案法を説明する．

[提案法]

Step 1: S_1-最遅排他的ノード (n_9) の上流に配置されている S_1-排他的ノードであるサニタイザ (n_5) の上流ノード (n_1, n_4) および下流ノード (n_9, n_{16}, n_{18}) を一

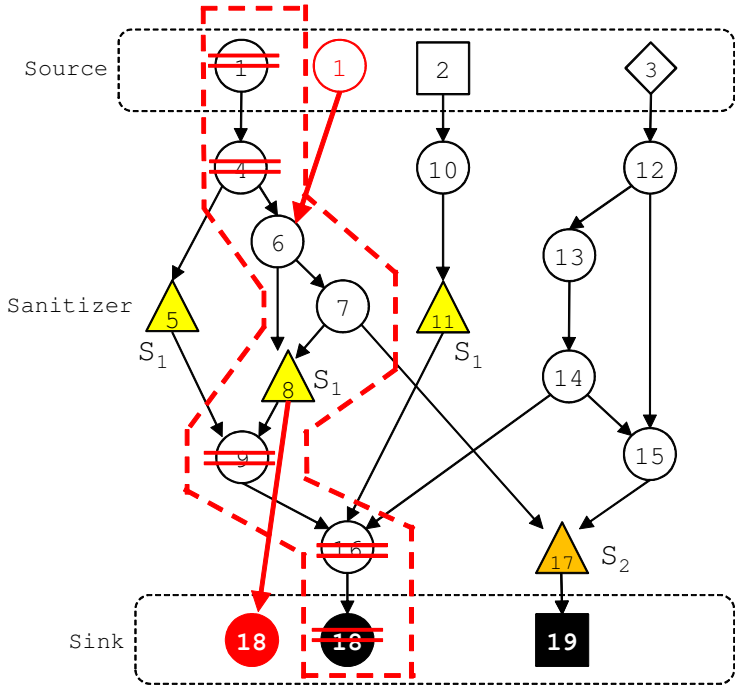

図 5 提案法の実施例
Fig. 5 An Example of the Proposed Method.

時的に削除する．

Step 2: 削除に伴いソースおよびシンクとの接続が切れたサブグラフができた場合，仮想的なソースノード (n_1) およびシンクノード (n_{18}) を設定し，それらとそのサブグラフを接続する．(仮想パス $n_1 \to n_6(\to n_7) \to n_8 \to n_{18}$ が構築される．)

Step 3: 仮想パスを含むデータフローグラフに対し，ノードベースアプローチを適用し，S_1-最遅排他的ノード (n_8) を求め，そこにサニタイザを配置する．

Step 4: 仮想的なソースノードおよびシンクノード，および仮想パスを削除し，Step 1 で一時的に削除したノードを元に戻す．

以上により，n_9 にサニタイザを配置した場合と同等の効果がある，n_5 および n_8 による妥当なサニタイザ配置が得られる．

5. 考察

本節では，提案法に対する妥当性，およびサニタイザの配置数に関する考察を述べる．

5.1 サニタイザの二重適用の回避

提案法では，サニタイザの二重適用は発生しない．なぜなら，提案法の Step 1 において，着目したサニタイザの上流および下流ノード，すなわち着目したサニタイザと二重適用が発生する可能性があるノードのすべてが一時的に削除され，選択候補から除外されるためである．

5.2 サニタイザの配置数

提案法で決定されるサニタイザ配置は，既存のサニタイザ配置を固定したとき存在し得るすべての妥当なサニタイザ配置の中で，サニタイザの配置数が最小である．なぜなら，**図 6** に示されるように，あるデータフローグラフからソース 1 個，シンク 1 個，およびそれらの間のすべてのパスのエッジを抽出したサブグラフを考えるとき，サニタイザがない状態では，シンクの直前のノードが最遅排他的ノードであり，ここにサニタイザを配置すれば，サニタイザの配置数は 1 となり，最小となる．しかし，図 6 に示される既存のサニタイザを残留させ，不足するサニタイザを追加する方針とした場合，提案法の Step 3 により，仮想サブグラフ G_1, G_2, および G_3 のそれぞれにおいて，最遅排他的ノードに必要なサニタイザが配置される．この配置は，明らかに，既存のサニタイザ配置を固定したとき存在し得るすべてのサニタイザ配置の中で，サニタイザの配置数は最小である．

上記は，局所的なサブグラフにおける議論であるが，一般のデータフローグラフの場合においても，同様に成り立つ．

6. 関連研究

サニタイザの妥当性に関する関連研究には以下がある．Balzarotti ら [1] は，静的および動的解析を組み合わせることにより，サニタイザの適切性を検査する方法を提案し，Saner と呼ばれるツールに実装している．Livshits ら [4] は，サニタイザ配置のための自動化技術を提案してい

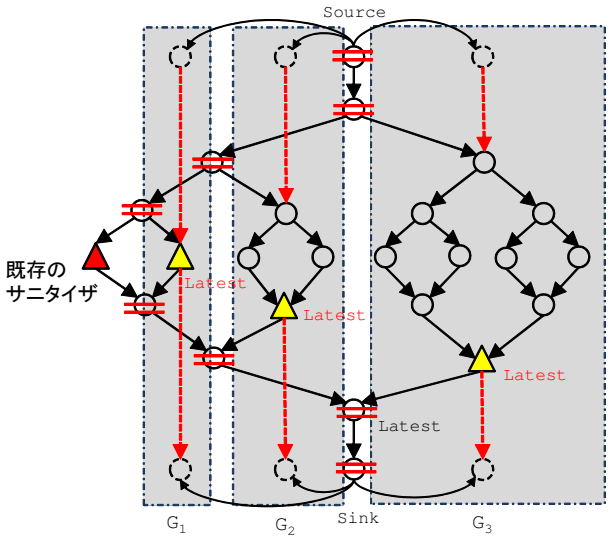

図 6 提案法における仮想サブグラフでの最遅排他的ノード計算の例
Fig. 6 An Example of Computation of Latest-Exclusive Nodes in Virtual Sub-Graphs in the Proposed Method.

る．彼らは，静的解析のみで解決できる場合は静的解析で行い，そうでない場合は動的解析で得られる情報を活用することにより，常に妥当なサニタイザ配置を得る方法を提案している．Hooimeijer ら [3] は JavaScript 等の汎用言語に正しいサニタイザを記述するためのドメイン固有言語 BEK を提案している．

静的解析の関連研究には以下がある．大浜ら [10] は，汚染解析を用いた静的解析ツールを提案し，実装評価を行っている．小黒ら [11] は，レースコンディション脆弱性に対し，Java バイトコードを解析することにより検出精度を高めた静的解析ツールを提案し，Eclipse[*3] プラグインに実装している．

7. まとめと今後の課題

本稿では，まず，現存する多くのセキュリティに関する静的解析ツールの機能が，汚染解析に基づいて脆弱性の可能性がある箇所を指摘するまでに留まっているため，対象プログラムに即した具体的な改修（対象プログラムにおけるサニタイザの適切な配置）は開発者が注意深く行う必要があり，これにかかるコストが大きいという背景を述べた．

次に，Livshits [4] らが形式化したサニタイザ配置問題の定義を導入し，彼らが提案したノードベースアプローチを紹介した上で，既に開発されたソフトウェアに対する改修を想定した場合は，開発プロジェクトの様々な価値基準により，同アプローチによる最適解が必ずしも最適とはならない場合があるという課題を示した．

次に，同課題を解決する一方法として，妥当ではないサニタイザ配置に対し，既存のサニタイザ配置を残留させながら，妥当なサニタイザ配置を与える方法を提案した．提案法は，実際のソフトウェア開発においてプログラムの改修および再試験などの様々なコストを勘案した場合に，開発者へ有用な改修方針を提示できる可能性を持つ．

今後取り組むべき課題は以下の通りである．

- フレームワークでよく用いられるリフレクション技術を用いたプログラムに対し，データフローグラフを正しく生成できる技術の開発
- 提案法を静的解析ツールに実装した上での性能評価

[*3] Eclipse は，米国およびその他の国における Eclipse Foundation, Inc. の商標または登録商標です．

参考文献

[1] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna., G.: *Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications*, In Proceedings of 2008 IEEE Symposium on Security and Privacy, pp. 387-401 (2008).

[2] Chin, E. and Wagner, D.: *Efficient Character-level Taint Tracking for Java*: In Proceedings of SWS'09: the 2009 ACM workshop on Secure web services, pp. 3-12 (2009).

[3] Hooimeijer, P., Livshits, B., Molnar, D., Saxena , P. and Veanes, M.: *Fast and Precise Sanitizer Analysis with BEK*, In Proceedings of the 20th USENIX conference on Security (2011).

[4] Livshits, B. and Chong, S.: *Towards Fully Automatic Placement of Security Sanitizers and Declassifiers*, In Proceedings of POPL 2013: 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 385-398 (2013).

[5] OWASP Top Ten Project, `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

[6] OWASP XSS Filter Evasion Cheat Sheet, `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`

[7] Samuel, M., Saxena, P. and Song, D.: *Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers*, In Proceedings of the 18th ACM conference on Computer and Communications Security, pp. 587-600 (2011).

[8] Saxena, P., Molnar, D. and Livshits, B.: ScriptGard: *Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications*, In Proceedings of the 18th ACM conference on Computer and Communications Security, pp. 601-614 (2011).

[9] Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R. and Song, D.: *A Systematic Analysis of XSS Sanitization in Web Application Frameworks*, In Proceedings of ESORICS 2011: the 16th European Symposium on Research in Computer Security, LNCS 6879, pp. 150-171 (2011).

[10] 大浜伸之，宮崎博：データフロー解析に基づくソースコード脆弱性検証手法，情報処理学会 コンピュータセキュリティシンポジウム 2007 (CSS2007) 論文集，pp.423-428 (2007).

[11] 小黒博昭，市原尚久，道坂修：Web アプリケーション脆弱性発見のためのソースコード診断ツールの開発，情報処理学会研究報告 2008-CSEC-41 (17), pp.97-102 (2008).