修士論文

静的値範囲解析による脆弱性検出手法

Static Range Analysis for Vulnerability Detection

指導教員　坂井 修一 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

初田 直也

概要

近年，コンピュータの普及・インターネットの発展に伴い，プログラムの脆弱性による被害は深刻なものとなっている．「プログラムに脆弱性がある」ということは，そのプログラムがシステムの乗っ取りや機密情報の奪取など，保安上の脅威となる行為に利用可能である，ということを意味する．この脆弱性はプログラムのバグが原因であることもあるが，バグのような明白な欠陥だけではなく，プログラム作成者が予想しなかった利用形態や設計段階での見落としなどが原因であることも多い．

一般に，脆弱性はその存在が確認されてからプログラムの修正作業を行うことになるが，その修正作業にはある程度の時間がかかり，その間プログラムの使用者は危険に晒されることになる．この問題を解決するためには，プログラムをリリースする時点で可能な限り脆弱性を発見・修正する必要がある．

本論文では，静的値範囲解析によってプログラムの脆弱性を検出する手法を提案する．脆弱性の原因はプログラム作成者がプログラムの動作の可能性を完全には把握できていない点にある．そのため，プログラム作成者に動作の可能性を提示することで，脆弱性を発見することができると考えられる．

提案手法は，静的値範囲解析と範囲付き命令の抽出の 2 つの手法から成る．静的値範囲解析は，プログラム上の値がとり得る範囲を静的な解析によって求める方法である．この静的値範囲解析によって得られた情報をそのままプログラム作成者に提示することも可能であるが，大量の情報から脆弱性に関連のあるものを発見することは容易ではない．そこで，範囲を持ったオペランドをとるような命令を抽出することで，脆弱性と関連の深い命令のみをプログラム作成者に提示する．これにより，効率の良い脆弱性の検出が可能となる．

テスト・プログラムを用いた評価を行った結果，いくつかの既知の脆弱性を検出することができた．また，従来の脆弱性検出手法では対象とされていない脆弱性が検出できる可能性を示した．さらに，提案手法における計算時間・メモリ消費について評価を行い，比較的大きなプログ

ラムに対しても適用可能であることを示した.

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

近年，コンピュータの普及・インターネットの発展に伴い，プログラムの脆弱性による被害は深刻なものとなっている.「プログラムに脆弱性がある」ということは，そのプログラムがシステムの乗っ取りや機密情報の奪取など，保安上の脅威となる行為に利用可能である，ということを意味する．この脆弱性はプログラムのバグが原因であることもあるが，バグのような明白な欠陥だけではなく，プログラム作成者が予想しなかった利用形態や設計段階での見落としなどが原因であることも多い.

一般に，脆弱性はその存在が確認されてからプログラムの修正作業を行うことになるが，その修正作業にはある程度の時間がかかり，その間プログラムの使用者は危険に晒されることになる．この問題を解決するためには，プログラムをリリースする時点で可能な限り脆弱性を発見・修正する必要がある.

1.2　研究の目的

本研究では，プログラムの動作の可能性に着目した脆弱性検出手法を提案する．提案手法では，

- プログラムの動作の可能性を解析する.
- 解析によって得られた動作の可能性から，脆弱性と関連の深い情報を抽出する.

の 2 段階に分けて脆弱性を検出する．さらに，それぞれを実現するために,「静的値範囲解析」及び「範囲付き命令の抽出」を提案する．静的値範囲解析では，プログラム上で値がとり得る範囲を静的に求めることで，プログラムがどのように動作する可能性があるかを明らかにする．また，範囲付き命令の抽出では，命令のオペランドが範囲を持つ場合に，その

命令が脆弱性と関連が深いことを利用し，そのような命令を抽出することで脆弱性を検出する．

1.3 論文の構成

本論文の構成は以下の通りである．まず，2 章ではこれまでに提案された脆弱性検出手法について述べ，その問題点を明らかにする．3 章では脆弱性の原因について考察した後に，提案手法の概要について述べる．4 章では静的値範囲解析について述べ，5 章では範囲付き命令の抽出について述べる．次に，6 章では実際に脆弱性を含んだテスト・プログラムを用いて検出を行い，その結果について議論する．最後に 7 章でまとめと今後の課題について述べる．

第2章　既存手法とその問題点

本章では，これまでに提案されたプログラムの脆弱性検出手法について述べ，その問題点を考察する．

2.1　既存の脆弱性検出手法

既存の脆弱性検出手法には，大きく分けて2つの方法がある．1つはプログラムのソース・コードなどから脆弱性の原因となり得る記述を検出する手法であり，もう1つはプログラムの実行中に脆弱性を利用した攻撃が行われたことを検出する手法である．

前者の例としては，C言語の標準ライブラリが提供する関数のうち脆弱性の原因となりやすいもの（gets()やstrcpy()など）の使用を検出する手法[5]や，C言語の文字列操作関数を対象に文字列の長さについての制約条件を生成し，文字列バッファのオーバーフローを検出する手法[7]が挙げられる．また，いくつかの脆弱性について，その脆弱性が検出されるようなassert文をソース・コードに挿入する手法も提案されている[2]．

後者の例としては，コンパイラによって攻撃を検出するコードを生成する手法やプロセッサ自体が攻撃を検出する手法が挙げられる．コンパイラを用いた方法としては，スタックにおけるバッファ・オーバーランの脆弱性に着目し，関数の戻りアドレスと共に乱数を格納し，関数からの復帰時にその乱数が変更されていないことを確認することで，戻りアドレスの正当性を確かめる，というものが提案されている[3]．また，バッファ・オーバーランなどの脆弱性において，入力によって与えられたコードが実行される点に着目し，プロセッサとオペレーティング・システムが協調して入力データを追跡し，入力データの実行を検出する手法が提案されている[6]．

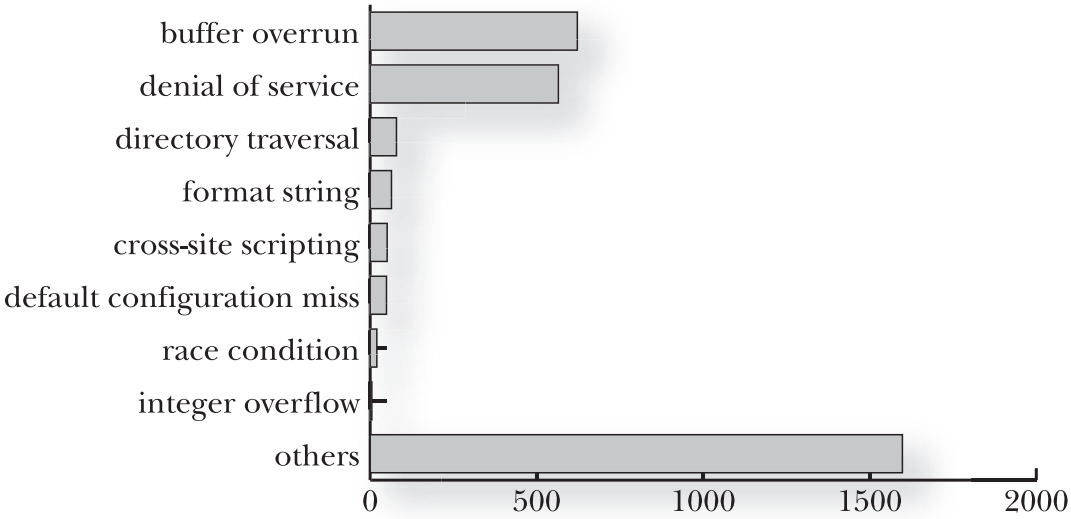

図 2.1: 脆弱性の報告件数

2.2 既存手法の問題点

既存手法の問題点の 1 つに，特定の脆弱性のみを対象としている点が挙げられる．図 2.1 は Common Vulnerabilities and Exposures[1] で報告されている脆弱性を分類したものである．多くの既存手法で対象となっているバッファ・オーバーランの脆弱性は確かに最も報告件数が多く，全体の約 20%を占めている．しかし，仮にバッファ・オーバーランの脆弱性を全て検出できるとしても，それは全体の 20%を検出したに過ぎず，残りの 80%については検出できない．さらに，脆弱性にはバッファ・オーバーランやフォーマット・ストリングのような分類に当てはまらないものが全体の約半分を占めている．この分類できない脆弱性はそれぞれ異なる脆弱性であり，それらに対し個別に検出手法を考案することは現実的ではない．また，今後新たに発見される脆弱性についても発見の度に個別に対応することになり，根本的な解決には至らない．

この問題を解決するには，特定の脆弱性を対象とせず，脆弱性の本質に基づいた検出手法が必要である．本論文では脆弱性の根本的な原因について考察し，それに基づいた脆弱性検出手法を提案する．

第3章　提案手法の概要

3.1　脆弱性の原因

プログラムの脆弱性とは，プログラムが作成者の意図しない形でシステムの乗っ取りやデータの改竄などに利用される，という欠陥である．脆弱性を利用された場合のプログラムの動作は，プログラム作成者にとって想定外であり，これを作成者自身が発見することは困難である．

プログラムは基本的には作成者の意図通りに書かれるはずであるが，想定外の挙動を示すことがある．その原因は，プログラムへ作成者にとって想定外の入力が行われるためである．例えば，人の名前を入力として受け付けるようなプログラムを考える．人の名前はアルファベットであるという想定に対して，制御文字という想定外の入力がなされる可能性がある．同様に，人の名前は高々100 文字程度であるという想定に対して，1000 文字の名前が入力される可能性もある．

このような想定外の入力に起因する想定外の挙動が脆弱性の本質である．

3.2　提案手法の概要

前節で述べたように，脆弱性が想定外の入力に起因するものであるなら，全ての入力を想定することで脆弱性を発見することが可能であると考えられる．すなわち，プログラムへの全ての入力を想定し，それに対する動作を列挙することで，プログラム作成者は列挙された動作の中から想定外であったものを発見することができる．

しかし，プログラムの動作が大量に列挙された場合，プログラム作成者がその動作を想定内・想定外と区別することは困難である．この問題を解決するためには，大量に列挙された動作から，プログラム作成者にとって想定外である可能性の高い動作を抽出すればよい．

本論文では以下のような流れでプログラムの脆弱性を検出する手法を提案する．

- 与えられたプログラムの動作として可能性のあるものを静的な解析により全て列挙する．
- 列挙された動作のうちプログラム作成者にとって想定外である可能性の高いものを抽出し，提示する
- 提示された情報に基づき，プログラム作成者が脆弱性の有無を判断する．

また，この検出手法のために以下の 2 つの手法を提案する．

- 静的値範囲解析
- 範囲付き命令の抽出

すなわち，静的値範囲解析によりプログラムの動作を解析し，範囲付き命令の抽出により想定外である可能性の高い動作を抽出する．

以下，4 章では静的値範囲解析について，5 章では範囲付き命令の抽出について述べる．

第4章　静的値範囲解析

本章では，プログラムの動作の可能性を解析する手法として提案した静的値範囲解析について述べる．

4.1　可能性の表現

3.2 節で，プログラムへの全ての入力を想定し，それに対する動作を列挙すると述べたが，文字通りに全てを列挙することは現実的ではない．そこで動作の可能性を表現する手段として，範囲の集合を用いる．通常プログラム上の変数は値が一意に定まっているが，これを範囲として拡張することでその値のとり得る可能性を表現することができる．

例えば，変数 a が 0 以上 20 以下，もしくは 30 以上 40 以下の値を取る可能性がある場合，以下のように表現する．

$$a = [0, 20][30, 40]$$

また，値が一意に決まる場合でも同様の表現が可能である．

$$a = [20, 20]$$

これ以降，全ての値は範囲の集合として扱う．

4.2　静的値範囲解析

本節では，プログラムを前節で定義した範囲の集合からなる方程式とみなし，プログラムから方程式を導く方法について述べる．

ここではプログラムを，基本ブロックとその接続関係からなる制御フローグラフを用いて表すことにする．

図 4.1 に 1 つの基本ブロックを示す．基本ブロックでは，制御がブロックの先頭に与えられた後に，ブロックの途中で停止したり，途中から分

岐したりせずにブロックの末尾から制御が離れる．従って，基本ブロック中の処理は単一の関数 $f(\boldsymbol{r})$ として表すことができる．$\boldsymbol{r}$ はプログラム中の全変数を要素とするベクトルであり，またその各要素は前節で定義したように範囲の集合である．図 4.1 に示すように，基本ブロックに入力された変数 $\boldsymbol{r_0}$ は基本ブロック中の処理 $f(\boldsymbol{r})$ により $\boldsymbol{r_1}$ として出力される．

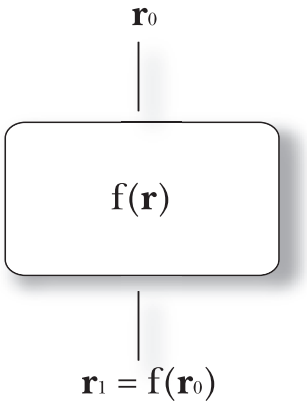

図 4.1: 基本ブロック

次に基本ブロック同士の接続関係について考える．基本ブロック同士の接続関係には分岐と合流の 2 種類がある．

まず，分岐の場合を図 4.2 に示す．分岐には分岐条件があり，これは分岐後のブロックに対する制約条件となる．図 4.2 では条件を $x \geq a$ $(x \in \boldsymbol{r})$，$x < a$ $(x \in \boldsymbol{r})$ としているため，分岐後の出力 $\boldsymbol{r_2}$，$\boldsymbol{r_3}$ にはそれぞれ条件が加えられている．

合流の場合は，プログラムの制御がいずれのパスを通ってくる可能性もあるため，各パスの出力の和集合を取る．図 4.3 では $f(\boldsymbol{r})$，$g(\boldsymbol{r})$ どちらのブロックも通る可能性があるため，$h(\boldsymbol{r})$ への入力は各出力の和集合である $f(\boldsymbol{r_0}) \cup g(\boldsymbol{r_1})$ となる．

4.3 現実的な処理

前節で述べた方法により，プログラムの制御フローグラフ全体について方程式を立て，それを解くことができれば全ての変数の範囲を求めることができる．しかし，任意のプログラムから生成される任意の方程式を解く方法は一般には存在しない．また，前節ではプログラムにおける演算のみに注目しており，メモリや入力の存在を考慮していない．以下

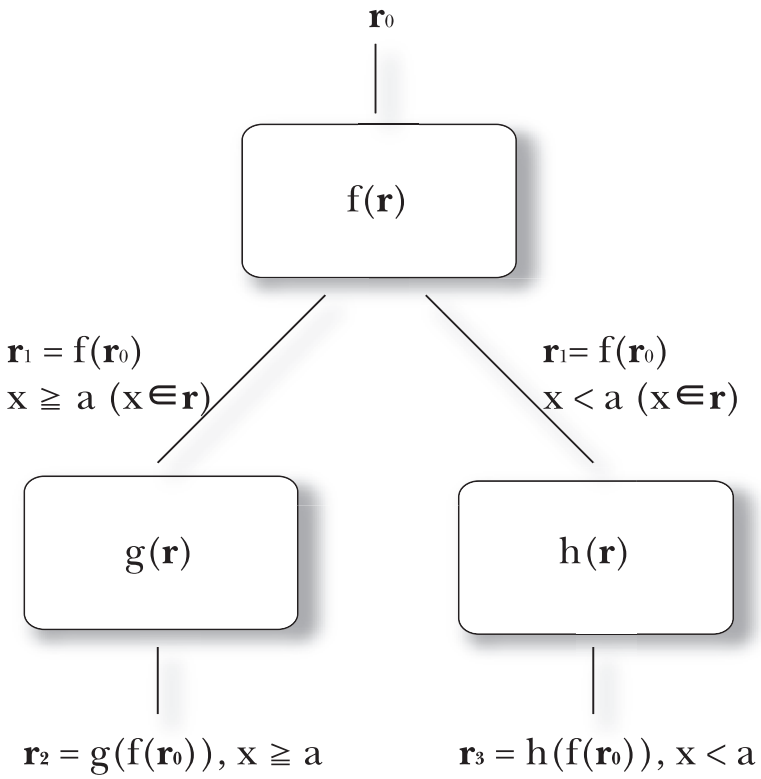

図 4.2: 基本ブロックの接続関係：分岐

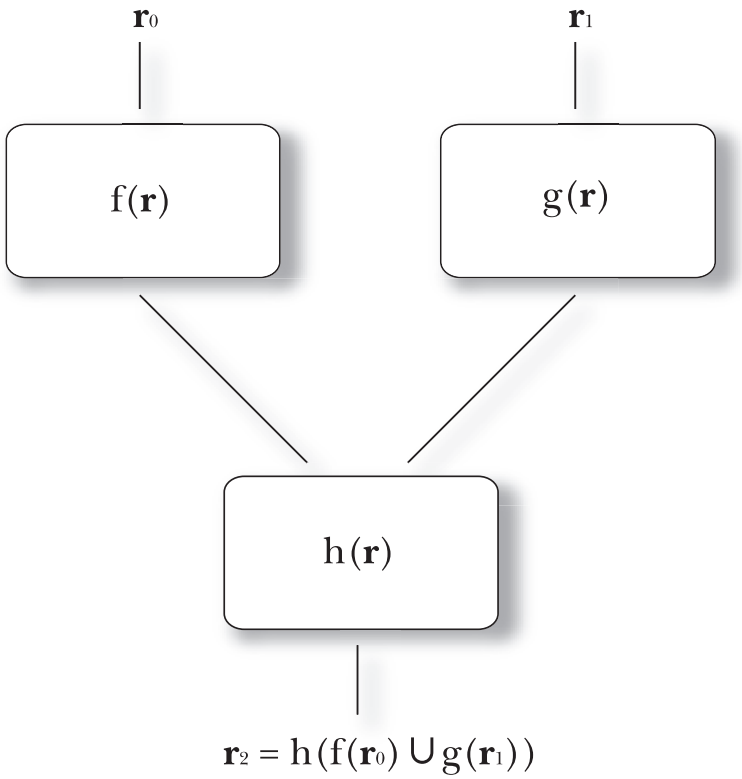

図 4.3: 基本ブロックの接続関係：合流

では，それら実際にプログラムを解析する際に考慮すべきいくつかの事項について述べる．

4.3.1 基本ブロックの処理

まず，基本ブロックにおける処理について述べる．4.2 節では基本ブロックを方程式として表したが，基本ブロックがループを構成しない場合にはその方程式を解く必要はない．図 4.1 の場合，入力の $\boldsymbol{r_0}$ に対し，基本ブロック中の演算を順に行うだけで $\boldsymbol{r_1}$ を得ることができる．

ただし，値を範囲の集合と定義したため，範囲の集合に対する演算を定義しなければならない．例えば，加算は以下のように定義できる．

$$[a, b] + [c, d] = [a + c, b + d]$$

範囲の集合に対する演算の詳細は付録 A にて述べる．

4.3.2 ループ

4.2 節で立てた方程式を解くことの難しさの 1 つにループがある．図 4.4 にループにおける方程式を示す．4.2 節で述べた分岐と合流の原理に従えば，$f(\boldsymbol{r})$ への入力は合流であるため $\boldsymbol{r_0} \cup g(f(\boldsymbol{r_0}))$ である．この入力は $f(\boldsymbol{r})$，$g(\boldsymbol{r})$ を通って再び $f(\boldsymbol{r})$ へ入力される．従って $f(\boldsymbol{r})$ への入力は

$$\boldsymbol{r_0} \cup g(f(\boldsymbol{r_0})) \cup g(f(g(f(\boldsymbol{r_0})))) \cup \cdots$$

となる．これは漸化式として

$$\boldsymbol{r_n} = \begin{cases} \boldsymbol{r_0}, & n = 0, \\ g(f(\boldsymbol{r_{n-1}})), & n \neq 0, \end{cases} n = [0, N-1].$$

のように表すことができる．この漸化式を解き，分岐の条件である $x < a$ $(x \in \boldsymbol{r})$ を満たすような N を求めれば，ループ脱出後の変数 $\boldsymbol{r_N}$ を求めることができる．N が定まればループ中の $\boldsymbol{r}$ の可能性である $\boldsymbol{r_n}, n = [0, N-1]$ も定まり，ループ内の変数の挙動が明らかになる．また，結果的に N はループの回数を示している．

このように，漸化式を解くことでループ中の変数を求める事ができるが，任意の漸化式を解くことは困難である．しかし，実際のプログラム

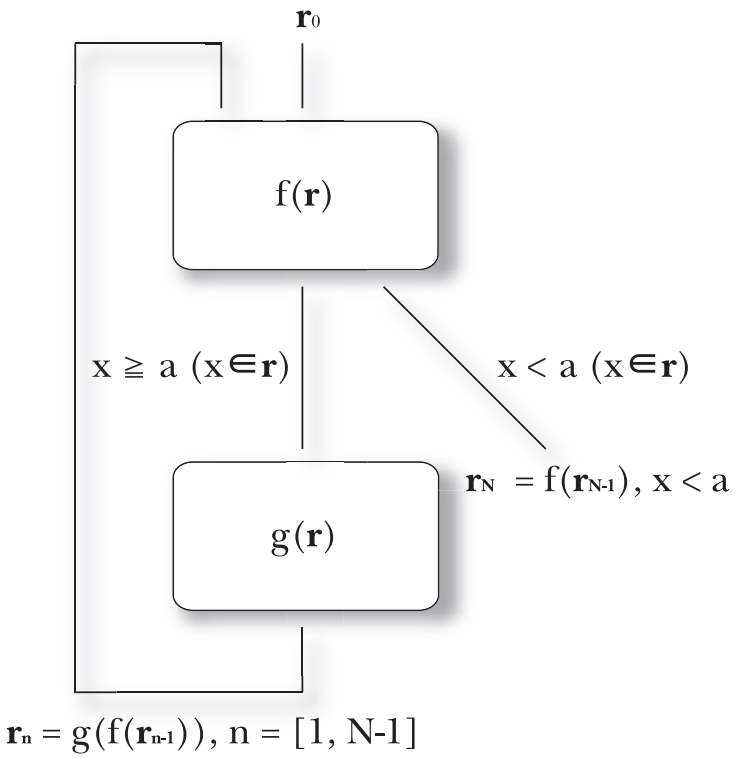

図 4.4: ループにおける方程式

では分岐条件にある変数 x が $x_n = x_{n-1} + 1$ のような単純な形をしていることが多い．このような場合に限れば，漸化式を解きループ回数である N を求めることができる．また，x を求めることができれば $y = x + 3$ のような関係にある変数 y は漸化式を解くことなく求めることができる．そこで，以下のような方法により，解けるものと解けないものを区別することを考える．

- 変数 $\boldsymbol{r}$ の要素のうち，ループ中で変更されるものを「不確定」，変更されないものを「確定」と区別する．
- 「不確定」である要素のうち，以下に述べる「簡単な漸化式」の条件を満たすものは「条件付確定」とする．
- 「不確定」である要素のうち,「確定」である要素と定数からなる演算と等式で結ばれているものは「確定」とする．
- 「不確定」である要素のうち,「確定」である要素,「条件付確定」である要素及び定数からなる演算と等式で結ばれているものは「条件付確定」とする．

ここで「条件付確定」とは，ループ回数が求まる場合に「確定」となることを意味する．従って，ループ回数を求める事ができれば「条件付確定」は「確定」に，求める事ができなければ「不確定」になる．

次に，上で述べた方法における「簡単な漸化式」について述べる．ここでは

$$x_n = ax_{n-1} + b$$

のように，漸化式が 1 変数かつ 1 次の場合を「簡単な漸化式」とする．この場合，漸化式は以下のように解くことができる．

$$x_n = \begin{cases} x_0 + bn, & a = 1, \\ a^n x_0 + \frac{b(1-a^n)}{1-a}, & a \neq 1. \end{cases}$$

最後に，ループにおける変数を求める方法について，その流れをまとめる．

- ループ中の変数を上記の方法に従い「確定」,「条件付確定」,「不確定」に分類する．
- 「条件付確定」である変数についての漸化式を解く．

- ループの脱出条件と解いた漸化式からループ回数 N を求める．
- 求めた N から「条件付確定」である変数を確定する．
- もし，N を求めることができなければ,「条件付確定」は「不確定」とする．

4.3.3 メモリ

プログラムの挙動を列挙する上で，メモリの存在は不可欠である．例えば buffer overrun の脆弱性は，関数の戻りアドレスと関数のローカル変数がメモリ上で近接して配置されている，という事実に基づいている．このような脆弱性の挙動を列挙するためには，メモリを含め，できる限り実際のプロセッサに近いモデルを用いる必要がある．

しかし，4.1 節で述べたように本手法では値を範囲の集合として定義しているため，範囲の集合をアドレスとするロードやストアを定義する必要がある．図 4.5 にアドレスが範囲の集合である場合のストアの様子を示す．一つ目の

store [20,30], ([1004,100c])

ではアドレスが範囲を持ち，1004 から 100c までのいずれのアドレスへもストアされる可能性がある．従って，1004 から 100c のデータはこれまで保持していたデータとストアするデータ [20,30] の和集合となる．二つ目の

store [12,13], ([1008,1008])

ではアドレスが 1008 と定まっているため，和集合を取る必要はなく，単に新しいデータで上書きすればよい．

一方，図 4.6 はアドレスが範囲の集合である場合のロードの様子を示している．ロードにおいてアドレスが範囲を持っている場合，その範囲のいずれのデータもロードされる可能性があるため，範囲内の全データの和集合がロードされる．

4.3.4 プログラムへの入力

プログラムにおいて，入力はその挙動に多様性を与える原因である．そのためプログラム内の演算のみではなく，その入力にもどのような可能性があるかを考慮しなければならない．

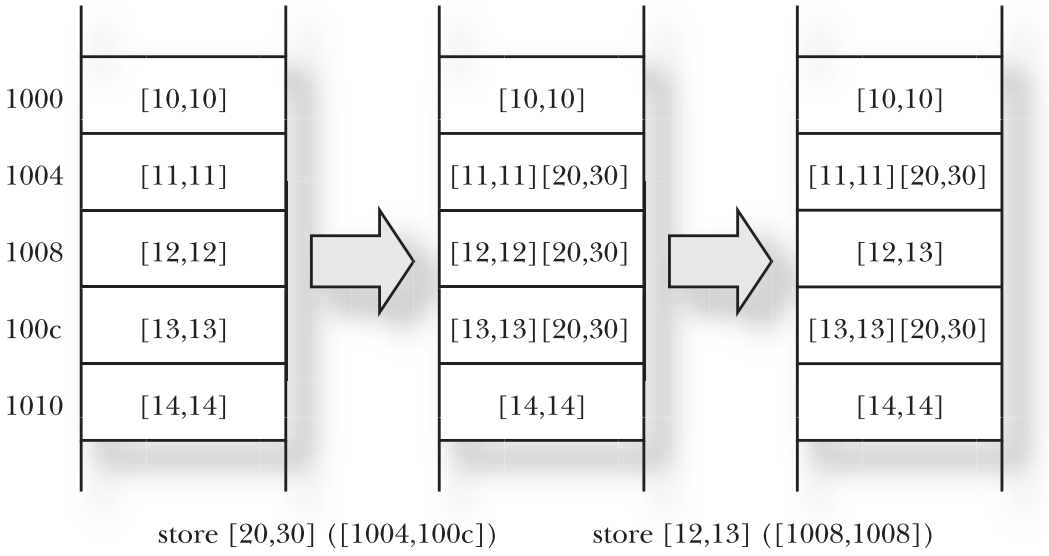

図 4.5: アドレスが範囲を持つ場合のストア

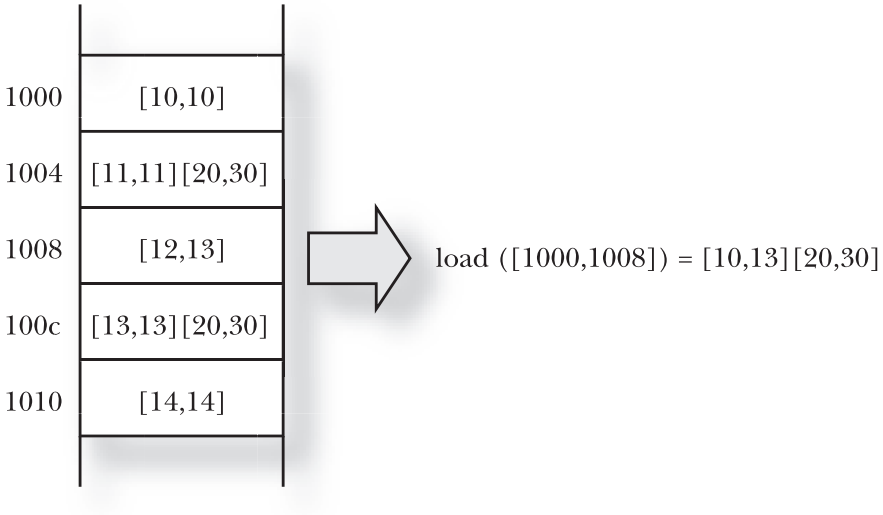

図 4.6: アドレスが範囲を持つ場合のロード

プログラムへの入力は，その入力のサイズが決まれば範囲を決めることができる．例えば，1 バイトの入力があればその可能性は [0,ff] である．また，入力はプログラム上ではオペレーティング・システム（OS）のシステム・コールとして捉えることができる．入力に対応するシステム・コールは read であり，これは入力を書き込むアドレスと書き込みサイズを指定して呼び出される．従って，read システム・コールがプログラムから呼ばれた場合，指定されたアドレスから指定されたサイズだけ [0,ff] をメモリに書き込むことで，任意の入力を表現できる．

4.4 誤検出の原因

本節では，提案する解析手法によって誤検出が発生する可能性について述べる．まず，4.3.2 節で述べたとおり，ループの解析において漸化式を解くことができない場合に一部の変数が「不確定」となる．本質的に「不確定」である変数と，解析の制限により「不確定」となるものは区別できないため，どちらも本質的に「不確定」と扱わざるを得ない．そのため本来脆弱性とは関係のない部分が脆弱であると検出されてしまう．

また，4.2 節で述べた分岐と合流の原理で「合流は和集合をとる」としたが，これは厳密には和集合とはならない．合流の各パスはそのいずれも通る可能性があるが，複数のパスを同時に通ることはない．しかし和集合をとった場合，ある変数はパス 1 を通り，またある変数はパス 2 を通った結果である，という状態が表現されてしまう．これを厳密に表現するためには，パス 1 を通った場合とパス 2 を通った場合を区別しなければならない．しかし，そのためには分岐ごとに状態数が倍に増えてしまい，現実的な時間とメモリで解析を行うことができなくなるため，提案手法では和集合とした．この結果，プログラムの挙動として本来ありえないものが列挙されてしまう可能性がある．

第5章　範囲付き命令の抽出

本章では，4章で述べた手法により得られた動作の可能性から，想定外である可能性の高いものを抽出する手法について述べる．さらに，具体的な脆弱性を例に挙げながら，その手法によって確かに脆弱性が検出されることを示す．

5.1　範囲付きの命令

4章で述べた手法は，範囲の集合としてプログラムの動作の可能性を得るものであった．すなわち，

store [20,30] ([1004,100c])

や

jmp ([2000,2010])

のように命令のオペランドが範囲の集合であることにより，その命令にどのようなオペランドが渡る可能性があるかを表現している．ここで挙げたように，オペランドが複数の可能性を持つ（$[10, 10]$ のように一意に定まらない）ような命令を「範囲付きの命令」と呼ぶことにする．

プログラムの動作から想定外である可能性の高い動作を抽出する方法を考えると，まず範囲付きでない命令は除外することができる．なぜなら，範囲付きでない命令はどのような入力に対しても常に同じオペランドを取り，想定外の動作をする可能性がないからである．

また，範囲付きの命令であっても演算命令は抽出する意味がない．これは，演算命令の結果が範囲付きだったとしても，その結果が演算命令以外の命令で使用されなかった場合には，プログラムの動作に影響を及ぼすことがないためである．

従って，プログラムの動作から想定外である可能性の高い動作を抽出する方法として，以下の3つの命令のうち範囲付きのものを抽出する，という方法を提案する．

- ジャンプ命令
- ロード・ストア命令
- システム・コール

以下では，主要な脆弱性について，本節で述べた方法によりそれらが検出できることを示す．

5.2 脆弱性の具体例

図 2.1 に示したように，最も多く報告されている脆弱性は buffer overrun であり全体の 20%に上る．以下，報告数の多い順に denial of service, directory traversal，format string となる．このうち denial of service についてはサービスを不正に停止することができる，という脆弱性であるため，プログラムによってその原因は様々であり，ここで取り上げる例としては適当でない．そこで，ここでは denial of service 以外の 3 つ，すなわち buffer overrun，directory traversal 及び format string を取り上げる．

5.2.1 Buffer Overrun

Buffer overrun とは確保したバッファのサイズより大きな入力があった際に，そのバッファに隣接したメモリ領域が入力によって上書きされ，任意のコードが実行される脆弱性である．

5.1 に最も一般的なスタックにおける buffer overrun の様子を示す．バッファをローカル変数として持つ関数が呼ばれたとき，スタックにはそのバッファと関数からの戻りアドレスが近接して配置される．さらに，通常戻りアドレスはバッファから見て上位アドレス側にあり，これはバッファの書き込み方向と等しい．従って，バッファのサイズより大きな入力があれば，その入力は戻りアドレスを上書きすることができる．そのため，バッファの先頭付近にコードを書き，戻りアドレスがバッファの先頭を指すように上書きすることで，任意のコードを実行することができる．

Buffer Overrun の検出　Buffer overrun において最も重要な点は，アドレスの上書きにより任意のアドレスへ制御が移る点である．4.3.4 節で述べた入力の扱いにより，入力で上書きされたアドレスは 64 ビットならば

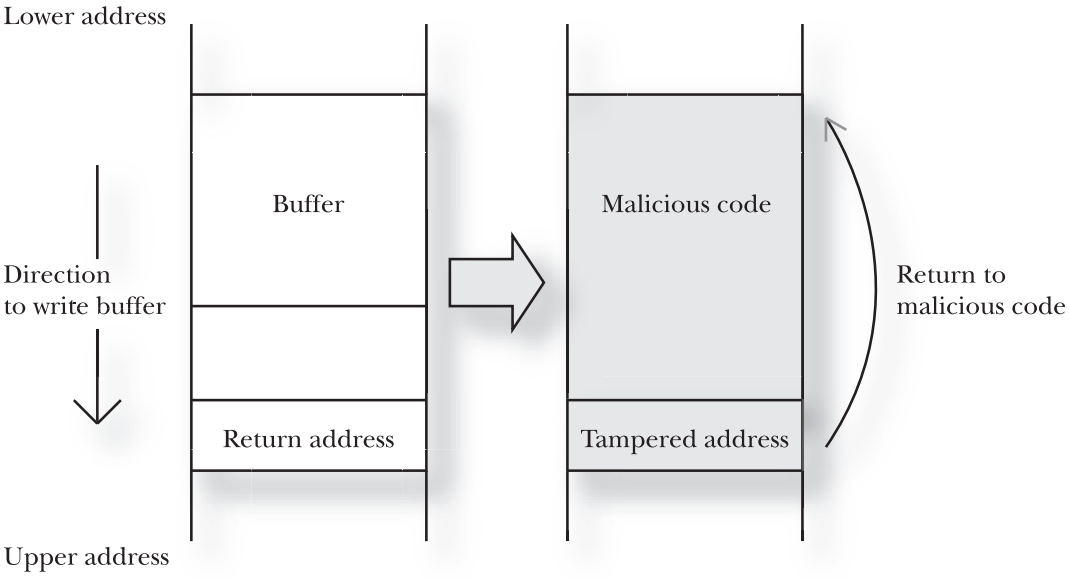

図 5.1: スタックにおける buffer overrun

[0, ffffffffffffff] となる．5.1 節で述べたように，これは範囲付きアドレスへのジャンプ命令として検出することができる．

ただし，テーブルジャンプや仮想関数呼び出しなど実行時にジャンプ先が決まるような場合は，buffer overrun と同様に範囲付きアドレスへのジャンプ命令として検出される．この判別は最終的にはプログラム作成者によって行われる必要があるが，テーブルジャンプや仮想関数呼び出しなどでは一般にアドレスの範囲は狭く，判別は容易である．

5.2.2 Format String

Format string の脆弱性とは，C 言語の標準出力関数である printf においてそのフォーマット文を入力から設定する場合，任意のアドレスへ任意のデータを書き込むことができる，というものである．

通常 printf 関数は

```
char buf[10];
scanf("%s", buf);
printf("%s\n", buf);
```

のように固定されたフォーマット文（この場合は%s\n）を引数に取る．しかし，

```
char buf[10];
scanf("%s", buf);
printf(buf);
```

のように，フォーマット文を変数とすることも可能である．この場合 buf は入力より任意の文字列を取りうるため，printf 関数は任意のフォーマットを解釈することになる．

一方，printf 関数のフォーマット指定子には%n がある．これはここまでに書き込んだ文字列のサイズを，対応する引数が指すアドレスに書き込むものである．さらに，%m$n とすれば m 番目の引数を使用すると明示的に示すことができる．従って，例えば buf に AAAA%m$n を入力すれば，printf 関数はスタックのトップから m 番目の領域に書かれたアドレスへ，AAAA のサイズである 4 を書き込む．このとき，buf に書かれた AAAA はスタック上にあるため，m を調整することで書き込むアドレスを AAAA（すなわち 0x41414141）とすることができる．また AAAA%1000A%2$n のようにフィールド幅を指定することで任意のデータ（この場合は 1004）を書き込むことができる．このように，format string の脆弱性では任意のアドレスに任意のデータを書き込むことができる．

Format String の検出　Format string の脆弱性の本質は，任意のアドレスへ任意のデータを書き込む点にある．そのため，これを検出するためにはメモリへの書き込みであるストア命令におけるアドレスとデータの範囲を確認すればよい．一般に配列への書き込みなど，範囲を持ったアドレスへのストアは多くあるが，配列の境界チェックなどが適切に行われていれば，そのアドレスの範囲は限定される．提案手法では，範囲付きアドレスへのストア命令を抽出することで，不自然に広い範囲のアドレス（例えば [0, ffffffffffffffff]）へのストアの有無を確認することができる．

5.2.3 Directory Traversal

Directory traversal とは，入力されたファイルへのパスなどに対して適切なチェックを行わなかったことにより，本来アクセスされるべきでないファ

イルへのアクセスが行われる，という脆弱性である．例えば，/home/hoge というディレクトリの下にあるファイルにユーザがファイル名を指定してアクセスする，という状況を考える．このとき，../../etc/passwd のようなファイル名を指定することで/home/hoge 以外のディレクトリへアクセスすることができる．この脆弱性を防ぐには，入力されたファイル名のチェックを行い，../のようなディレクトリを遡る記述がないことを確認すればよい．

Directory Traversal の検出　Directory traversal のような脆弱性は，そのプログラムにどのようなファイルを開く可能性があるか，という問題である．従って，ファイルを開くためのシステム・コールである open にどのようなファイル名が渡される可能性があるかを調べればよい．また，open と同様に write システム・コールも，プログラムの外部に影響を及ぼすという点で，注意が必要である．特にユーザからの入力が open や write に渡る際には，適切なチェックが行われているかどうかを確認する必要がある．提案手法では，open や write に渡る情報の可能性を得ることができるため，この確認を比較的容易に行うことができる．

第6章　テスト・プログラムによる評価

本節では，提案手法の実装とそれを用いた脆弱性の検出について述べる．実装した検出ツールでは，命令セットとして Alpha 21264[4] を想定し，バイナリを入力に取る．ツールは入力されたバイナリを解析し，範囲を持ったアドレスやデータを引数に取るジャンプ命令，ロード・ストア命令及びシステム・コールがあれば，それらを危険な命令として出力する．

以下では，入力したテスト・プログラムの詳細と，検出結果について述べる．

6.1　Buffer Overrun

Buffer overrun については stack におけるものと heap におけるものの 2 種類を用意した．

6.1.1　Stack における Buffer Overrun

スタックにおける buffer overrun の原理は第 5.2.1 節で述べたとおりである．以下に，スタックにおける buffer overrun の脆弱性を持つテスト・プログラムを示す．

```
int test(){
  char buf[10];
  read(0,buf,100);
  return 0;
}
int main(){
```

```
  test();
  return 0;
}
```

このプログラムでは，read 関数によって 100 バイト読み込むため配列 buf がオーバーフローし，test 関数の戻りアドレスを上書きする．

このプログラムを検出ツールに入力した結果が以下である．

```
ranged jump target: [0, fffffffffffffc] at 120003184
```

これはジャンプ命令のターゲット・アドレスが [0, fffffffffffffc] であることを示している．また，120003184 は命令のアドレスを示しており，ここでは関数からの return にあたる．

6.1.2 Heap における Buffer Overrun

Heap における buffer overrun は一般に，stack におけるものよりも困難である．Stack では戻りアドレスの上書きによって任意のアドレスへ制御を移すことが可能であったが，heap に戻りアドレスはない．以下に，heap における buffer overrun の一例を示す．

```
class base{
protected:
  char buf[10];
public:
  virtual ~base(void){return;};
  virtual void rd(void){return};
};
class test: public base{
public:
  virtual ~test(void){return;}
  void rd(void){
    read(0,buf,100);
    return;
```

```
  }
};
int main(){
  base *t = new test();
  t->rd();
  delete t;
  return 0;
}
```

このプログラムでは new 演算子により，heap にクラス test の領域を確保している．クラス test はクラス base を継承し，仮想関数を含んでいるためその内部に仮想関数テーブルへのポインタを保持している．クラスのメンバ関数を呼ぶ際には，このポインタから仮想関数テーブルを参照し，ジャンプ先のアドレスを決定する．

この仮想関数テーブルへのポインタは，コンパイラによってはメモリ上でメンバ変数である buf の上位側に配置される．このとき，buf をオーバーフローすれば仮想関数テーブルへのポインタを上書きすることができる．上書きしたポインタの先に，実行したいコードを指すポインタのテーブルを用意しておけば，任意のコードを実行することができる．上記のプログラムでは，delete t によってデストラクタが呼ばれる際に，任意のコードが実行される．

このプログラムを検出ツールに入力した結果が以下である．

```
ranged jump target: [0, fffffffffffffffc] at 12000b96c
```

結果は stack における buffer overrun と同じく，範囲を持つアドレスへのジャンプが検出されている．

6.2 Format String

Format string の脆弱性は第 5.2.2 節で述べたとおりである．printf 関数のフォーマット文を変数とすることにより発生するため，テスト・プログラムは以下のようになる．

```
int main(){
  char buf[10];
  read(0,buf,10);
  printf(buf);
  return 0;
}
```

このプログラムを検出ツールに入力した結果の抜粋が以下である．

```
unfixed data is stored to unfixed address at 12000b0c4
unfixed data is stored to unfixed address at 12000b0cc
unfixed data is stored to unfixed address at 12000b0dc
```

これはストア命令のアドレスとデータが第 4.3.2 節で述べた「不確定」であることを示している．printf 関数はフォーマット文の処理が無限ループとして構成されており，ループの脱出条件は次に処理するフォーマット文が NULL 文字であることである．従って，第 4.3.2 節で述べた方法ではループの回数を求める事ができず，多くの変数が「不確定」となっている．

本来，format string の脆弱性として検出されるべきストア命令は 10 箇所あるが，実際にはその 10 箇所を含む計 57 箇所のストア命令が検出された．すなわち，再現率は 100%，適合率は 17.5%である．

6.3 Directory Traversal

Directory traversal の脆弱性は本来，プログラムの仕様と密接に関連しているため，実際に使用されているプログラムに対して検出を試みるべきである．しかし，本論文における実装では検出したアドレスとソース・コードとの対応が取れないため，大規模なプログラムの検証は困難である．そこでここでは open システム・コールについて，以下のプログラムを用いて引数の可能性が示されることを確かめる．

```
int main(){
```

```
    char buf[5];
    read(0,buf,5);
    open(buf);
    return 0;
  }
```

このプログラムを検出ツールに入力した結果が以下である．

```
open system call:
[0, ff], [0, ff], [0, ff], [0, ff], [0, ff]
```

プログラムでは 5 文字のファイル名を読んでファイルを開いているため，5 つの [0, ff] が open されるファイルの可能性として示されている．

6.4 計算量の評価

本節では，前節までで用いた 3 種のテスト・プログラムについて，解析にかかる時間とメモリ使用量を評価する．表 6.1 は各プログラムのバイナリ・サイズと解析にかかった時間、メモリの使用量を示している．ただし，各バイナリは静的にライブラリをリンクしている．

表 6.1: 実行時間・メモリ消費量

	サイズ [KB]	解析時間 [s]	メモリ消費量 [MB]
Buffer overrun	144	0.13	9.96
Format string	192	7.76	88.2
Directory traversal	144	0.12	9.98

Buffer overrun と directory traversal は解析時間・メモリ使用量共に少ないが，これはバイナリの大部分が静的にリンクされたが実際には使用されていない関数であるためである．Format string では printf 関数を使用しており，この関数のサイズに比例して時間・メモリ使用量が増えていると考えられる．

また，提案手法では 1 命令に対して定数回の処理しか行わないため，解析時間のオーダはプログラムのサイズに対して O(n) である．従って，上に挙げたテスト・プログラムより大きなプログラムであっても現実的な時間で解析可能である．

6.5 評価のまとめ

本章の評価では，buffer overrun の脆弱性を実際に検出できることを示した．一方で，format string における評価で見られたように，ループにおける解析に失敗した場合に誤検出が発生している．また，directory traversal のように，従来の手法では検出の対象と考えられていなかった脆弱性であっても，提案手法によって検出できる可能性があることを示した．

また，計算時間・メモリ消費は対象とするプログラムのサイズに対し O(n) のオーダであり，大きなプログラムであっても現実的な時間で解析が可能であることを示した．

第7章　おわりに

7.1　本論文のまとめ

本論文では，プログラムの動作の可能性に着目したプログラムの脆弱性検出手法を提案した．脆弱性の原因はプログラム作成者がプログラムの動作の可能性を完全には把握できていない点にある．そのため，プログラム作成者に動作の可能性を提示することで，脆弱性を発見することができると考えられる．

提案した検出手法は，静的値範囲解析と範囲付き命令の抽出の 2 つの手法から成る．静的値範囲解析は，プログラム上の値がとり得る範囲を静的な解析によって求める方法である．この静的値範囲解析によって得られた情報をそのままプログラム作成者に提示することも可能であるが，大量の情報から脆弱性に関連のあるものを発見することは容易ではない．そこで，範囲を持ったオペランドをとるような命令を抽出することで，脆弱性と関連の深い命令のみをプログラム作成者に提示する．これにより，効率の良い脆弱性の検出が可能となった．

テスト・プログラムを用いた評価を行った結果，いくつかの既知の脆弱性を検出することができた．また，従来の脆弱性検出手法では対象とされていない脆弱性が検出できる可能性を示した．さらに，提案手法における計算時間・メモリ消費について評価を行い，比較的大きなプログラムに対しても適用可能であることを示した．

7.2　今後の課題

今後の課題については以下のものが挙げられる．

- 検出精度の向上

  6 章における format string の脆弱性検出で見られたように，複雑なプログラムにおいては誤検出が発生する．これは主に 4.4 節で述べ

た原因に因るものであるが，より複雑な漸化式への対応などによりある程度の改善が可能であると考えられる．

- ユーザ・インターフェースの開発

  提案手法ではプログラム作成者に情報を提示するためのユーザ・インターフェースが非常に重要である．提案手法ではプロセッサに近い実行モデルを採用したことにより，プロセッサの動作を理解していないプログラム作成者にとっては理解できない情報が提示されることになる．この情報を適切に変換し，プログラム作成者が理解し易い形で提示するユーザ・インターフェースについても今後検討する必要がある．

参考文献

[1] Common Vulnerabilities and Exposures. `http://www.cve.mitre.org/`.

[2] B. V. Chess. Improving Computer Security using Extended Static Checking. In *Proc of. 2002 IEEE Symposium on Security and Privacy*, pages 160–173, 2002.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc of. 7th USENIX Security Conference*, pages 63–78, 1998.

[4] R. E. Kessler. The Alpha 21264 Microprocessor. In *International Symposium on Microarchitecture*, 1999.

[5] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proc of. 10th USENIX Security Conference*, pages 177–190, 2001.

[6] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proc of. 2004 International Symposium on Architectural Support for Parallel Languages and Operating Systems*, pages 85–96, 2004.

[7] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc of. 7th Network and Distributed System Security Symposium*, pages 3–17, 2000.

発表文献

主著論文

- Bus Serialization for Reducing Power Consumption
  初田 直也, 卒業論文, 東京大学 工学部 (Feb. 2004)

- Bus Serialization for Reducing Power Consumption
  Naoya Hatta, Niko Demus Barli, Chitaka Iwama, Luong Dinh Hung, Daisuke Tashiro, Shuichi Sakai, and Hidehiko Tanaka
  情報処理学会 研究会報告 ARC 2004-ARC-159(Jul 2004)

- Bus Serialization for Reducing Power Consumption
  Naoya Hatta, Niko Demus Barli, Chitaka Iwama, Luong Dinh Hung, Daisuke Tashiro, Shuichi Sakai, and Hidehiko Tanaka
  情報処理学会 ACS 論文誌, Vol.47, SIG3(ACS13), (Mar. 2006)(掲載予定)

- 静的値範囲解析によるプログラムの脆弱性検出手法
  初田 直也, 入江 英嗣, 五島 正裕, 坂井 修一
  先進的計算基盤システムシンポジウム SACSIS 2006, (May. 2006)(投稿中)

共著論文

- Steering and Forwarding Techniques for Reducing Memory Communication on a Clustered Microarchitecture
  Hidetsugu Irie, Naoya Hattori, Masanori Takada, Naoya Hatta, Takashi Toyoshima and Shuichi Sakai
  International Workshop on Innovative Architecture 2005(IWIA2005), (Jan. 2005)

- Distributed Speculative Memory Forwarding
  Hidetsugu Irie, Naoya Hattori, Masanori Takada, Naoya Hatta, Takashi Toyoshima and Shuichi Sakai
  IEEE Symposium on Low-Power and High-Speed Chips(COOL Chips VIII), (Apr. 2005)

謝辞

本研究を進めるにあたり，指導教官である坂井修一教授には学部 4 年からの 3 年間に渡って多くのご指導，ご助言を頂きました．また，五島正裕助教授からも，1 年間と短い期間ではありましたが，大変多くのご指導を頂きました．ここに深く感謝の意を表します．

清水修助手，八木原晴水さん，月村美和さん，北原杏さんには，研究室における設備の導入や各種事務手続きなど，研究室で過ごすための様々なご支援を頂きました．

入江英嗣博士，豊島隆志氏，清水一人氏，栗田弘之氏をはじめ，D グループのメンバーの皆様には，ミーティングにおける議論を通して，貴重なご意見を頂きました．

研究室の計算機，及びネットワークの構築・管理のために，Luong Dinh Hung 氏，豊島隆志氏，清水一人氏，栗田弘之氏には多大なご尽力を賜りました．心より感謝致します．

本研究の一部は，21 世紀 COE「情報技術戦略コア」，及び科学技術振興機構 CREST「ディペンダブル情報基盤」によります．

付 録A　範囲の集合に対する演算

本章では，範囲の集合に対する演算の定義を行う．本論文における実装では以下のように算術演算，論理演算を定義した．

$$
\begin{aligned}
\text{加算} &: [a,b]+[c,d]=[a+c,b+d] \\
\text{減算} &: [a,b]-[c,d]=[a-d,b-c] \\
\text{乗算} &: [a,b]\times[c,d]=[a\times c,b\times d] \\
\text{除算} &: [a,b]\div[c,d]=[a\div d,b\div c] \\
\text{剰余算} &: [a,b]\%[c,d]=[a\%c,b\%d] \\
\text{論理積} &: [a,b]\&[c,d]=[a\&c,b\&d] \\
\text{論理和} &: [a,b]|[c,d]=[a|b,c|d] \\
\text{左ビットシフト} &: [a,b]<<[c,d]=[a<<c,b<<d] \\
\text{右ビットシフト} &: [a,b]>>[c,d]=[a>>d,b>>c]
\end{aligned}
$$

上で定義した演算のうち，加算，減算，除算，右ビットシフトについてはこの定義は論理的に正しい定義である．しかし，それ以外の定義は不完全なものとなっている．乗算，左ビットシフトについては，この定義では演算結果として正しくないものが演算結果に含まれてしまう．例えば，乗算の定義に従えば

$$[10,20]\times[2,3]=[20,60]$$

となるが，実際には [20, 60] に含まれる値のうち 21，22 などは乗算の結果としては正しくない．

また，剰余算，論理積，論理和についてはそれぞれ結果として得られた範囲の最大値，最小値が，本当の最大値，最小値であるという保証はない．

これらの不完全な演算の定義は，誤検出の原因ともなるが，実際には演算子のオペランドの両方が範囲を持つことは少なく，本論文における誤検出には影響していない．