Modular product program を用いた情報漏洩検証の自動化

指導教員：

早稲田大学　基幹理工学部

情報理工学科　寺内研究室

1w163079-9

趙　元瑋

2020 年 02 月 03 日

概要

modular product program[1] という非干渉性を検証するための新しい手法が近年に提案された。これは、フロクラムコピーと自身との合成により問題を到達可能性検証の問題に帰着する方法である。ただし、合成の仕方が工夫されており、 既存の到達可能性検証手法で解きやすい形にできると主張されている。しかし、[1]の手法では不変条件等をユーザーが記述する必要のある手動/半自動的な検証枠組みでのみ使用しており、ソフトウェアモデル検査など自動到達可能性検証アルゴリズムと組み合わせてうまく使える手法なのかは未知数である。

この論文では、modular product program による手法が自動到達可能性検証アルゴリズムと組み合わせて使ったとき、効果的な結果を得られるかを調査する。modular product program の変換により得られたプログラムをそのまま自動到達可能性検証ツールに投入することにより試す。

目次

第一章　研究背景・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 4

1.1 プログラム検証・・・・・・・・・・・・・・・・・・・・・・・・ 4

1.2 非干渉・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 5

1.3 研究目的・・・・・・・・・・・・・・・・・・・・・・・・・・・ 6

第二章　関連研究・・・・・・・・・・・・・・・・・・・・・・・・・・・・7

2.1　Hyperproperty・・・・・・・・・・・・・・・・・・・・・・・・7

2.2　Modular product program・・・・・・・・・・・・・・・・・・・ 9

第三章　研究内容・・・・・・・・・・・・・・・・・・・・・・・・・・・14

3.1 プログラム検証機・・・・・・・・・・・・・・・・・・・・・・14

3.2　Modular product program について・・・・・・・・・・・・・・14

3.3 非干渉に満たすCプログラム・・・・・・・・・・・・・・・・・17

3.4 実験結果・・・・・・・・・・・・・・・・・・・・・・・・・・19

第四章　結論・・・・・・・・・・・・・・・・・・・・・・・・・・・・21

第五章　今後の課題・・・・・・・・・・・・・・・・・・・・・・・・22

第一章 研究背景

1.1 プログラム検証

プログラム検証とは，プログラムが何らかの性質を満たすことを証明することである。

近年，コンピュータは社会基盤として広く普及しており，その上で動作するプログラムに問題が生じると，たとえば自動改札機が開かなくなったり，株の売買が停止したり，ロケットが爆発したり，書きかけの論文が跡形もなく消えてしまったりと，われわれの生活に深刻な影響を及ぼす。

このようなプログラムの問題に対する従来の対策は、プログラムをテストして、プログラムに問題がないことを確認することである。しかしテストではプログラムの問題を見つけることはできるが、プログラムに問題がないことを証明することはできない。そのため、プログラム検証を処理する人の観点から、ゲタを飛ばして占うのに近いところがある。

これに対しプログラム検証では，数理論理学的手法に基づいてプログラムの性質を証明するため，たとえば「A という問題が生じないこと」という性質を検証すれば， A という問題が絶対に生じないことを保証できる。

プログラム検証を行う手法は二種類ある。ひとつは手動で行う方法，もうひとつは自動で行う方法である。

手動で検証を行う方法では，プログラムが何らかの性質を満たすことを人間が証明する。もちろん，プログラムに「私が証明しました」という顔写真付きのシールを貼っただけでは誰も信用しないので，定理証明支援系プログラムなど，証明の正しさを検証できるプログラムに入力として与えられる形式で証明を行うのが一般的である。

自動で検証を行う方法では，ある特定の性質に関して，もしくは入力として与えられた性質に関して，検証を行うプログラムを用いて証明する。たとえば Java や O'Caml のような静的に強く型付けされたプログラミング言語で

は，不正なメモリ操作をしないという性質などを，型理論にもとづいた型検査プログラムが自動的に証明する。また，ソフトウェアモデル検査手法では，プログラムが取り得る状態を網羅的に探索し，与えられた性質が満たされていることを自動的に証明する。

1.2 非干渉

プログラムが非干渉であるとは、そのプログラムが全く情報を漏洩しないことを指す。漏洩というのは、低いセキュリティーレベルの情報（例えば、公開されている情報）から、高いセキュリティーレベルの情報を得る、もしくは推測できることである。

例えば、下記の図1のパスワードの例は、それぞれpasswdという機密情報を扱うプログラムである。そのうち、Cは全く機密を外部に流出しないのに対し、Bは機密を完全に流出し、Aは passwdとユーザーが入力したguessが一致したかという情報のみ流出する。 よって、情報漏洩量は B＞A＞C となり、Cは非干渉であり、 AとBは非干渉ではない。

if (passwd = guess)
{
output("Yes")
} else {
output("No")
}
	if (passwd = guess)
	{
	output(passwd)
	} else {
	output(passwd)
	}
		if (passwd = guess)
		{
		output("Yes")
		} else {
		output("Yes")
		}

A	B	C

図1、パスワードの例

非干渉の検証というのは、あるプログラムMに対して、プログラムM'を作

る。

$\forall h_1 \in HighInputs, \forall h_2 \in HighInputs, \forall l \in LowInputs,$

M'(h_1, h_2, l):

$$O_1 = M_1(h_1, l)$$

$$O_2 = M_2(h_2, l)$$

$$assert\ (O_1 = O_2)$$

全てのh_1、h_2が高いセキュリティ入力にしており、lが低いセキュリティ入力にする時、M'が任意のh_1、h_2、lに対して、この式が成り立つ。つまり、assert errorが起こらないことである。

1.3 研究目的

本研究では、modular product program[1]の手法を用いて、プログラム検証機を使って、情報漏洩検証の自動化を実現することを目的としている。

本論文では、第二章にて関連研究の Hyperproperty と Modular product program について説明し、第三章で Modular product program を用いて、検証する内容や動作、実験結果について説明している。第四章では、本研究の結論を述べ、第五章では本研究を踏まえた今後の課題を述べる。

第二章 関連研究

2.1 hyperproperty [2]

Hyperpropertyを紹介する時、まずtracesというのはプログラムステップの有限また無限の集合である。そして、抽象的に状態の集合をΣにする。その時、Σ^*は、Σ上のすべての有限列の集合を示し、Σ^ωは、Σ上のすべての無限列の集合を示す。

$$\Psi_{fin} \triangleq \Sigma^*,$$

$$\Psi_{inf} \triangleq \Sigma^\omega,$$

$$\Psi \triangleq \Psi_{fin} \cup \Psi_{inf},$$

Trace propertyとは無限のトレースの集合である、式と表すと

$$Prop \triangleq \mathcal{P}(\Psi_{inf})$$

$\mathcal{P}$は冪集合。

すべてのプロパティは、safety propertyとliveness propertyの共通部分である。safety propertyとは、「悪いこと」を禁止すること。つまり、プログラムの実行中に予期外の状態に入ることはない（例えば、パラメータの不正な呼び出しなどの実行中のエラーなど）。

Pottier と Simonet は[4]、プログラムの 2 つの実行についての同時推論に基づいてセキュリティ情報流を検証するための型システムを開発した。Darvas

等の人たちは[5]、セキュリティ情報流を動的論理で表現できることを示す。 Barthe 等の人たちは[6]自己構成の構成に基づいて、Hoare 論理と時相論理の同等の定式化を行う。

Pの自己構成をプログラムP; P 'として定義する。P 'はプログラムPを示すが、すべての変数は新しい変数に名前が変更される。たとえば、変数xはx 'に名前が変更される。 Pがセキュリティ情報流を示すことを確認する1つの方法は、変換されたプログラムP; P 'の次のtrace propertyを確立することである。

すべてのlow変数lに対して、実行前にl = l 'が成り立つ場合、実行が終了しても、high変数の値に関係なく、l = l'が成り立つ。

hyperpropeertyを使って、上記の結果がより一般的な定理の特殊なケースであることを示すことができる。 k-safety hyperpropeertyを、kを超えるトレースが悪いことには決して関与しないsafety hyperpropertyとして定義する。

定義：hyperproperty Sがk-safety hyperpropeertyである場合には、

$$\forall T \in Prop: T \notin S \Longrightarrow (\exists M \in Obs: M \leq T \wedge |M| \leq k \wedge (\forall T' \in Prop: M \leq T' \Longrightarrow T' \notin S))$$

（$Obs \triangleq \mathcal{P}^{fin}(\Psi_{fin})$，ここで、$\mathcal{P}^{fin}(X)$はセットXのすべての有限サブセットの集合を示す。トレースセットのプレフィックス≤は、次のように定義される

$$T \leq T' \triangleq (\forall t \in T: (\exists t' \in T': t \leq t'))$$

)

2.2 Modular product program [1]

modular product program [1] という非干渉性を検証するための新しい手法が提案された。これは、プログラムコピーと自身との合成により問題を到達可能性検証の問題に帰着する方法である。k-safety hyperpropertyについてのモジュラー推論を可能にする新しい形のproduct programである。

```
procedure main (people)
            return (count)
{
    i := 0;
    count := 0;
    while(i<|people|){
        current := people[i];
        f := is_female(current);
        count := count + f;
        i := i + 1;
    }
}
procedure is_female (person)
            return (res)
{
    //gender encoded in first bit
    gender := person mod 2;
    if (gender == 0){
        res := 1;
    }else{
        res := 0;
    }
}
```

図 2、サンプルプログラム。 パラメーターpeople には、それぞれが人物の属性をエンコードする一連の整数が含まれている。このシーケンスの女性の数をカウントするプログラムである。

例から考えてみると、図2の方が一連の人々の中で、女性のエントリの数をカウントするプログラムである。明らかにこのプログラムが決定的である

ことがわかる。つまり、出力状態が入力によって完全に決定されることである。

そこで、この決定性を証明したい場合、下の図3のように、2-safety hyperpropertyを利用して、modular 2-productのプログラムに変換し、一回の実行で元のプログラムの2回実行が実現できるようにする。そして、同じ入力に対しては常に同じ結果を返すことにより、決定的なプログラムであることが証明できる。

```
procedure main (p1,p2,people1,people2)
            return (count1, count2)
{
    if(p1){ i1 := 0; }
    if(p2){ i2 := 0; }
    if(p1){ count1 := 0; }
    if(p2){ count2 := 0; }
    while((p1 && i1<|people|) || (p2 && i2<|people|){
        l1 := p1 && i1 < |people1|;
        l2 := p2 && i2 < |people2|;
        if(l1){ current1 := people1[i1]; }
        if(l2){ current2 := people2[i2]; }
        if(l1 || l2){
            t1, t2 := is_female(l1, l2, current1, current2);
        }
        if(l1){f1 := t1;}
        if(l2){f2 := t2;}
        if(l1){count1 := count1 + f1;}
        if(l2){count2 := count2 + f2;}
        if(l1){i1 := i1 + 1;}
        if(l2){i2 := i2 + 1;}

    }
}
procedure is_female (p1, p2, person1, person2)
            return (res1,res2)
{
    if(p1){
        gender1 := person1 mod 2;
```

```
    }
    if(p2){
        gender2 := person2 mod 2;
    }
    t1 := p1 && gender1 == 0;
    t2 := p2 && gender2 == 0;
    f1 := p1 && !(gender1 == 0);
    f2 := p2 && !(gender2 == 0);
    if (t1){ res1 := 1; }
    if (t2){ res2 := 1; }
    if (f1){ res1 := 0; }
    if (f2){ res2 := 0; }
}
```

図 3、図 2 のプログラムを modular 2-product を用いて、変換されたプログラム

Modular product program は、実行ごとに、現在の実行条件を保存するブールアクティベーション変数(boolean activation variables)を使用する。最初の全てのアクティベーション変数は真にする。

図 3 は図 2 の modular 2-product program である。最初に main procedure を考える。そのパラメーターは複製され、すべての変数が 2 つにコピーされ、それぞれ実行に 1 つある。これは、自己構成または既存のプロダクトプログラムに類似している。

さらに、変換された procedure には 2 つのブールパラメータ p1 と p2 がある。これらの変数は、 procedure の初期アクティベーション変数である。main はプログラムのエントリポイントであるため、初期アクティベーション変数は true であると想定できる。

people1 および people2 の任意の入力値でプロダクトプロダクトを実行するとどうなるかを考える。プロダクトは最初に i1 と i2 をゼロに初期化する。こ

れは、元のプログラムの i と同様である、また count1 と count2 についても同じである。

元のプログラムのループは、プロダクトの単一ループに変換された。元のループ条件がアクティブな実行に対して真である場合、その条件は真である。これは、元のプログラムが少なくとも 1 回実行される限り、ループが繰り返されることを意味する。

ループの中では、フレッシュアクティベーション変数 l1 と l2 は、対応する実行がループ本体を実行するかどうかを表す。つまり、実行ごとに、前のアクティベーション変数（p1 または p2）が真の場合、それぞれの起動変数は真になる。ループ内のすべてのステートメントは、これらの新しいアクティベーション変数を使用して変換される。したがって、少なくとも 1 回の実行でループが実行されている間、ループは繰り返し実行されますが、ループガードが実行に対して false になるとすぐに、その起動変数は false になり、ループ本体は無効になる。

変換された is_female は、条件式がどのように処理されるかを示している。実行ごとに 2 つずつ、4 つの新しいアクティブ化変数 t1、t2、f1、および f2 を導入し、最初のペア（t1,t2）は then 分岐で 2 つの実行のどちらかによって実行されるべきかどうかをエンコードする。 2 つ目は、else-branch で同じものをエンコードする。次に、これらのアクティベーション変数を使用して分岐を変換する。したがって、どちらの分岐も非アクティブな実行に影響を与えず、そして、アクティブな実行ごとに影響する。

要約すると、アクティベーション変数は、各実行で実行される状態変化ステートメントのシーケンスがプロダクトと元のプログラムで同じであることを保証する。

第三章　研究内容

3.1 プログラム検証機

今回使われたプログラム検証機は、「Ultimate Automizer」[3]というソフトウェア検証へのオートマトン理論的アプローチに基づいた safety properties の検証機である。この検証機を使って、assert を宣言して、assertion error が起こすかチェックする。通過する場合には、「assertion always hold」というメーセージが出る。

3.2 modular product program について

[1]では、図 2 と図 3 のプログラムを例として、modular product program である手法に対して、紹介されている。そこで、図 2 の方は元のプログラムで、図 3 の方が、modular product program を用いて、変換されたプログラムである。しかし、今回使ったプログラム検証機には、検証する時 C 言語であることを要求されているため、まず、例のプログラムを C 言語に書き換える必要がある。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TRUE 1
#define FALSE 0
#define MAX_LENGTH 60000 //  メモリ容量を制限するため
typedef int Bool;
typedef struct {
    int a;
```

```
    int b;
} Result;

Result is_female(Bool p1, Bool p2, int person1, int person2) {
    int res1, res2;
    int gender1, gender2;
    if(p1){
        gender1 = person1 % 2;
    }
    Bool t1 = p1 && gender1 == 0;
    Bool f1 = p1 && !(gender1 == 0);
    if (t1) {res1 = 1;}
    if (f1) {res1 = 0;}

    if(p2){
        gender2 = person2 % 2;
    }
    Bool t2 = p2 && gender2 == 0;
    Bool f2 = p2 && !(gender2 == 0);
    if (t2) {res2 = 1;}
    if (f2) {res2 = 0;}

    return (Result){res1, res2};
}

int main() {
    int i1 = 0, count1 = 0, i2 = 0, count2 = 0;
    int current1, current2;
    int t1, t2;
    int f1, f2;
    Bool p1 = TRUE;
    Bool p2 = TRUE;
    int length1 = (int)rand()%MAX_LENGTH;
    int people1[length1];
    srand((unsigned)time(NULL));
    if(p1){
        i1 = 0, count1 = 0;
    }
    int length2 = (int)rand()%MAX_LENGTH;
    int people2[length2];
    for (int i = 0; i < length1; i++) {
        people1[i] = rand()%2;
    }
    for (int i = 0; i < length2; i++) {
```

```
    people2[i] = rand()%2;
  }

  if(p2){
    i2 = 0, count2 = 0;
  }

  while ((p1 && i1 < length1) || (p2 && i2 < length2)) {
    Bool l1 = p1 && i1 < length1;
    Bool l2 = p2 && i2 < length2;
    if(l1){
      current1 = people1[i1];
    }
    if(l2){
      current2 = people2[i2];
    }
    if(l1 || l2){
      Result temp = is_female(l1, l2, current1, current2);
      t1 = temp.a;
      t2 = temp.b;
    }
    if(l1){
      f1 = t1;
      count1 += f1;
      i1 += 1;
    }
    if(l2){
      f2 = t2;
      count2 += f2;
      i2 += 1;
    }
  }
  if (count1 != count2) {
    //@ assert count1 != count2;
  } else {
    //@ assert count1 == count2;
  }
  return 0;
}
```

図 4、図 3 により、C 言語で書き換えたプログラム

そして、このプログラムがある場合には、違う入力に対して、同じ結果を返し、非干渉を満たすことを証明したいため、assert を宣言する。

3.3 非干渉に満たすCプログラム

今回では、非干渉に満たすCプログラムを下の図5のように、書いてみる。まずは、絶対値を求め、そして、出力に対して、大なりイコール0の場合は「yes」、そうではないとき「no」というようなプログラムである。絶対値を既に求めた上で、必ず「yes」と出力するため、このプログラムが明らかに非干渉を満たすプログラムである。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int abs(int b){
        if (b >= 0){
                return b;
        }else{
                return -b;
        }
}
int main(){
        int b = 0;
        int a = 0;
        int N = 0;
        int i = 0;
        srand((unsigned)time(NULL));
        b=rand()%5;
        N=rand()%10;
        a = abs(b);
        while(i<N){
                a+=a;
                i++;
        }
```

```
    if(a>=0) {
        printf("yes¥n");
    }else{
        printf("no¥n");
    }
    return 0;
}
```

図 5、非干渉に満たす C プログラム

また、図 5 のプログラムを基づいて、modular product program の方法を用いて、modular 2-product のプログラムを作る。下の図 6 に示したように、assert を宣言し、全部「yes」と出力することが、a1 と a2 が全部大なりイコール 0 と同値であるので、検証のために「a1*a2 > 0 ‖ a1*a2 == 0」と使って、検証条件として、プログラムをプログラム検証機に投入する。

```
#include <stdlib.h>
#include <time.h>

typedef int Bool;
#define TRUE 1
#define FALSE 0

typedef struct{
   int m;
   int n;
} Result;

Result abs_(Bool p1, Bool p2, int b1, int b2){
    int res1,res2;
    p1 = TRUE;
    Bool t1 = p1 && b1 >= 0;
    Bool f1 = p1 && !(b1 >= 0);
    if(t1) {res1 = b1;}
    if(f1) {res1 = -b1;}

    p2 = TRUE;
    Bool t2 = p2 && (b2 >= 0);
    Bool f2 = p2 && !(b2 >= 0);
    if(t2) {res2 = b2;}
```

```
    if(f2) {res2 = -b2;}

    return (Result){res1, res2};
}

int main(){
    int i1 = 0, i2 = 0, b1 = 0, b2 = 0;
    int a1 = 0, a2 = 0, N1 = 0, N2 = 0;
    Bool p1 = TRUE;
    Bool p2 = TRUE;
    srand((unsigned)time(NULL));
    b1 = rand()%5;
    b2 = rand()%5;
    if(p1 || p2){
        Result temp = abs_(p1, p2, b1, b2);
        a1 = temp.m;
        a2 = temp.n;
    }
    while((p1 && i1 < N1) || (p2 && i2 < N2)){
        Bool l1 = p1 && i1 < N1;
        Bool l2 = p2 && i2 < N2;
        if(l1){
            a1+=a1;
            i1++;
        }
        if(l2){
            a2+=a2;
            i2++;
        }
    }
    //@ assert    a1*a2 > 0 || a1*a2 == 0;
  return 0;
}
```

図 6、図 5 により変換された modular 2-product のプログラム

3.4 結果

図 6 をプログラム検証機に投入した結果は、図 7 に示したように、「assertion always hold」というメーセージが出るため、検証が成功し

た。

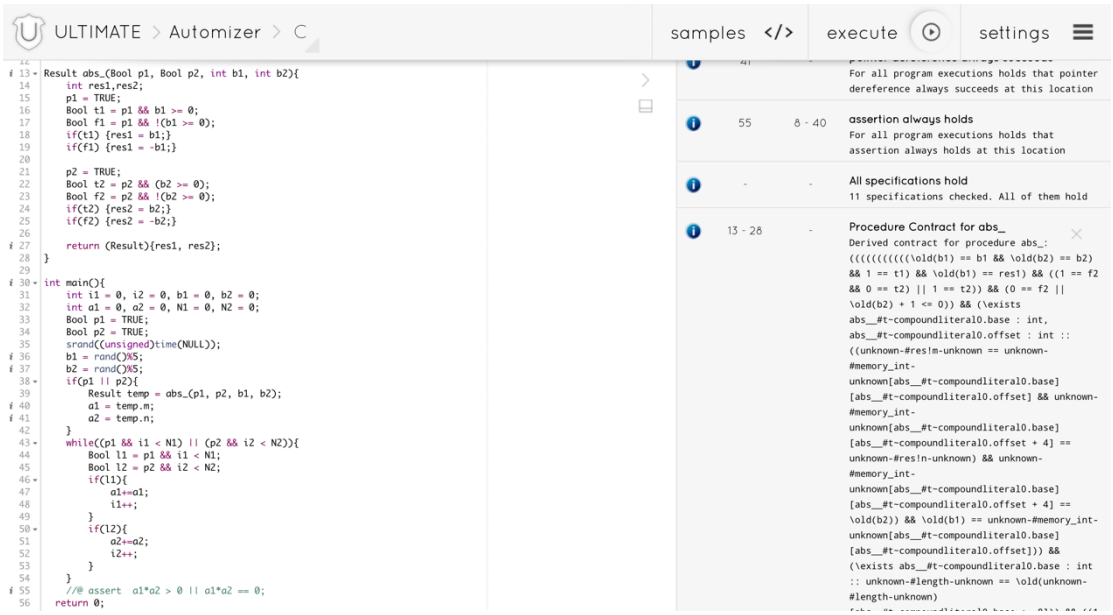

図 7、図 6 のプログラムがプログラム検証機での実行結果

第四章　結論

図 7 に示したように、図 6 のコードがプログラム検証機に投入することによる情報漏洩検証の自動化を成功した。その一方で、図 4 のコードは、プログラム検証機で検証する時、実行時間が長すぎで結果が出られなくなった。その原因を考えると、一つは、modular product program という方法には、まだ改良できることが一つで、もう一つは、プログラム検証機自身がまだ改善できると考える。

第五章　今後の課題

今回の実験では、すべての検証用のプログラムがmodular product programを用いて、手書きで変換されたプログラムにより実現するため、完全な自動化とは言えない。そのプログラム変換器に工夫する必要がある。また、本実験では、結論に言ったように、まだ不足点があるため、その不足点により改良することも今後の課題になると考える。

参考文献

[1] M. Eilers, P. Mu ¨ller, and S. Hitz. Modular product programs. In ESOP, 2018.

[2] Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. 2010

[3] The Ultimate Automizer.https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer

[4] F. Pottier and V. Simonet, Information flow inference for ML, in: *Proc. of ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002, pp. 319–330.

[5] Á. Darvas, R. Hähnle and D. Sands, A theorem proving approach to analysis of secure information flow, in: *Security in Pervasive Computing*, Lecture Notes in Computer Science, Vol. 3450, Springer, Berlin, 2005, pp. 193–209.

[6] G. Barthe, P.R. D'Argenio and T. Rezk, Secure information flow by self-composition, in: *Proc. of IEEE Computer Security Foundations Workshop*, Pacific Grove, CA, June 2004, pp. 100–114.

謝辞

本学士論文の作成にあたり，日頃よりご指導頂いた早稲田大学基幹理工学部の寺内多智弘教授に深く感謝をいたします．また，本研究を進めるにあたりご協力頂いた寺内研究室の皆さまに多大なる感謝をいたします。