解析対象への前提知識を必要としないバイナリプログラム解析技術

3.1 はじめに

本稿ではIIJ-II技術研究所で行っている、解析対象に対しての前提知識を必要としないバイナリプログラム解析技術について解説します。

プログラム解析とは、対象となるプログラムがどのような振る舞いを行うのかを分析する技法で、大別すると実際にプログラムを実行してその挙動を調べる「動的解析」と、プログラムを実行せずにプログラムの構成を解析する「静的解析」に分けられます。

動的解析の例としては、プログラムの各機能の整合性をチェックする単体テストや、ランダム生成された入力データを与えて挙動をテストするファジングツールなどがあります。静的解析の例としては、不要となる処理を排除したり、事前に計算可能な処理を行ったりといったやり方で実行時の計算処理効率を高める最適化解析や、プログラムがデータを意図しない扱いをすることで実行時に起こるエラーを回避するために、データ型の整合性を保証する静的型検査などがあります。動的静的を問わず、こうした様々なプログラム解析技術は統合開発環境(IDE)に組み込まれて、プログラム開発の効率化やバグの削減などに役立っています。

このようにプログラム解析技術は、主に開発する側が使用することが前提となっています。一方で、既に作成され配布されたプログラム(バイナリプログラム)に対して、その挙動の解析が求められる場合があります。例えば、マルウェアの疑いのあるプログラムの振る舞いを知りたい場合や、サードパーティから提供されたファームウェアが不審な挙動を起こさないか調べたい場合などが挙げられます。このような場合、対象となるプログラムのソースコードや、どのようなコンパイラを用いて作成されたかといった情報が、解析する側に必ずしも与えられているとは限りません。そのような場合でも動的解析は可能です。例えば隔離されたサンドボックス環境でマルウェアの疑いのあるプログラムを実行し、その挙動を解析する検疫システムなどが実用的に運用されています。しかし、動的解析の問題点として、解析できるのは実際に実行された制御パスだけである、ということが挙げられます。そのため、実行環境を判定して振る舞いを変更するアンチ解析マルウェアや、特定の入力に応じて挙動を変更するバックドアが仕込まれたファームウェアなどの解析は動的解析だけでは難しいと言えます。

従って網羅的に挙動を調べるためには、動的解析だけではなくバイナリプログラムを対象とした静的解析が必要とされています。バイナリプログラムの静的解析では、そのプログラムがどのように作成されたかなどの前提知識がどの程度与えられているかによって、その困難さが変わってきます。例えば、バイナリプログラムを作成するのに使用されたコンパイラが推定できる場合、そのコンパイラが生成するコードパターンに基づいて、元のソースコードを復元できる場合があります。このような手法を「逆コンパイル」と呼びます。逆コンパイルができれば、復元されたソースコードに既存の静的解析手法を適用することで、プログラムの振る舞いを解析することができます。

しかし、このような前提知識が常に得られる、あるいは、得られたとしても信用できるとは限りません。本研究ではそのような前提知識をほとんど得られない、つまり「得体の知れない」バイナリプログラムであっても解析が行える静的解析技術の開発を行っています。

次節以降では静的解析でバイナリプログラムを対象とすることの難しさ、そして前提知識の有無によって、なぜ更に困難さが増すのかを解説します。

3.2 バイナリプログラム解析

C/C++などのプログラミング言語で記述されたプログラム(「ソースコード」と呼ばれます)は、コンパイラと呼ばれるソフトウェアによって、CPUが処理できる単純な機械命令の列に変換されます。この機械命令の列は、CPUによって定められた符号化方法によって整数値バイトデータへとエンコードされます。こうしてバイトデータの列として表現されたプログラムを「バイナリコード」と呼びます。

作成されたバイナリコードは「実行ファイル」と呼ばれるファイル形式の中に埋め込まれています。実行ファイルにはバイナリコードの他に、実行時にメモリのどの位置にプログラムを配置するか、配置されたプログラムのどこから実行を開始(エントリアドレス)するか、そして、実行中にどのような外部ライブラリ関数を呼び出すか、などのメタ情報も記述されています。このような実行ファイルを本稿ではバイナリプログラムとして扱います。ソースコードを用いずバイナリプログラムのみを用いて行う静的解析を「バイナリコード解析」と呼びます。

身近な例で言えば、標準的なウイルス検知ソフトウェアでは、既知のマルウェアからシグネチャと呼ばれる特徴的なバイトデータ列を抽出したデータベースを利用して、バイナリコード内に既知のシグネチャが含まれているかどうかを判定しています。近年では実行ファイルに含まれるその他のメタ情報を含めて機械学習を行い、未知のマルウェアの検出を行うシステムもあります。

このようにバイナリコードを単なるバイトデータとして解析する手法はマルウェア検知のようにプログラムの自動分類を行うことには適していますが、プログラムがどのような挙動をするか詳細に知りたい場合にはバイナリコードを単なるデータ列としてではなく、プログラムとして解析する必要があります。このような解析を本稿では「バイナリプログラム解析」と呼びます。

バイナリプログラム解析ではバイナリコードからプログラムの制御構造を抽出して再構成する必要があります。そのための第一歩として「逆アセンブラ」という手法があります(図-1)。前述のように、各機械命令はアーキテクチャごとに定められた規則に従いバイトデータに変換されますが、これを逆向きに行い、バイトデータから機械命令の表現へ変換することを逆アセンブルと言います。

単純な逆アセンブラではバイナリコードの先頭から順次逆アセンブルする「線形走査型」という手法が利用されます。線形走査型の場合、途中にプログラムではないデータが混入していると、その時点から正確な逆アセンブラができなくなる可能性があります。一方、IDA Proなどの高度な逆アセンブラでは、エントリアドレスから再帰的に直接ジャンプ命令(ジャンプの行き先アドレスが命令中に実値で指定されている制御命令)を辿る「再帰探索型」という手法が用いられています(図-2)。再帰探索型の逆アセンブラでは、間接ジャンプ命令(ジャンプの行き先アドレスがレジスタやメモリに格納されている制御命令)に行き当たった場合は、それ以上の探索を行いません。これは一般的に間接ジャンプ命令の行先アドレスを静的に解決することが決定不能問題であり理論的に困難なためです。

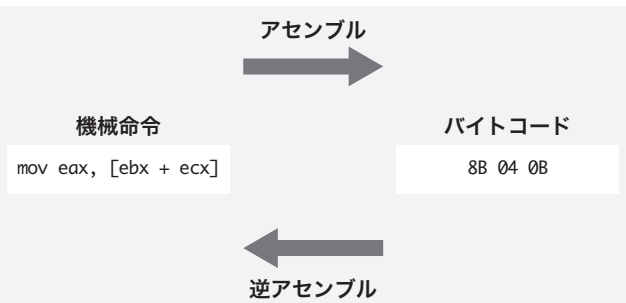

図-1 逆アセンブラ

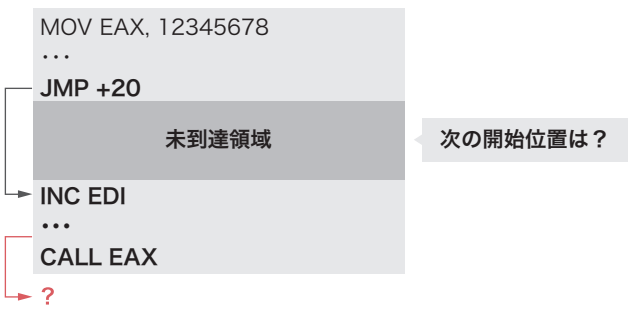

図-2 再帰探索型逆アセンブラ

よって、再帰的探索では直接ジャンプ命令だけでは辿れない未到達領域ができます。IDA Proなどの逆アセンブラは様々なヒューリスティクスを用いて、この未到達領域から適切な開始位置を推定して再帰的探索を再開します。そのようなヒューリスティクスの1つに「関数位置同定」と呼ばれる手法があります。

一般的にプログラム開発においては、「関数」もしくは「手続き」と呼ばれる独立した機能に分割して組み合わせることで、再利用性を高め、開発効率を高めることが行われています。各関数にどのように引数(パラメータ)を渡し、また計算結果を返り値としてどのように受け取るかは、CPUやオペレーションシステムなどによって「呼出規約」として定められています。内部関数がコンパイルされる場合には、この呼出規約に応じて必要な前処理と後処理が挿入されます。これらの処理はコンパイラによって固有のパターンがあるので、このパターンを見つけ出すことで、内部関数の位置を推定することができます。このような解析を関数位置同定と呼びます。

関数位置同定によってプログラムを関数単位に分割し、関数ごとに再帰探索を行うことで、プログラム全体の逆アセンブルが行えます(図-3)。

関数位置同定によって関数の開始位置を特定するためには、プログラムが呼出規約に従ってコンパイラで作成されていることが前提となっています。この前提が満たされない場合、逆アセンブラでは正確なプログラムの構造が再構成できなくなります。また、たとえコンパイラで生成されたプログラムであっても、強い最適化を行うと関数内の前処理や後処理のコードパターンが省略され、関数位置の同定が困難になる場合があることが報告されています[*1]。

逆アセンブラだけでプログラムの構造を再構成できない理由の1つは間接ジャンプ命令です。間接ジャンプ命令の行先アドレスを、静的データ解析を用いてできる限り静的に解決する解析手法を「制御フロー再構成」と呼びます。前述のとおり、関節ジャンプ命令の行先を静的に解決することは決定不能問題なので、完全解は得られません。CodeSurfer/x86[*2]やJakstab[*3]などの先行研究では抽象解釈(Abstract interpretation)と呼ばれる解析手法を用いて行先アドレスの近似解を求めています。

しかし、制御フロー再構成にはもう1つ困難な点があります。前述のとおり、プログラムは複数の関数によって構成されています。事前に関数位置同定が行われてプログラムが関数単位に分割されている場合は、関数ごとに独立して解析を行う手続内プログラム解析(Intra-procedural program analysis)を適用できます。十分な前提知識がなく関数位置同定が正確に行えない場合は、プログラム全体を解析する全体プログラム解析(Whole program analysis)を行うことになります。事前に関数位置の同定はできていなくても実際にはプログラムはいくつかの関数に分割されています。そのため、全体プログラム解析では「文脈依存性」を考慮する必要があります。

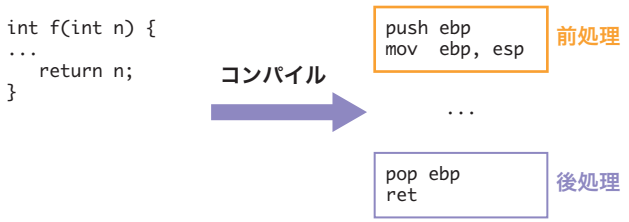

図-3 関数の前処理／後処理

*1 Andriesse2016:Andriesse, Dennis, et al. “An in-depth analysis of disassembly on full-scale x86/x64 binaries.” 25th {USENIX} Security Symposium ({USENIX} Security 16). 2016.
*2 Balakrishnan2005: Balakrishnan, G., Gruian, R., Reps, T., & Teitelbaum, T. (2005, April). CodeSurfer/x86—A platform for analyzing x86 executables. In International Conference on Compiler Construction (pp. 250-254). Springer, Berlin, Heidelberg.
*3 Kinder2008:Kinder, J., & Veith, H. (2008, July). Jakstab: A static analysis platform for binaries. In International Conference on Computer Aided Verification (pp. 423-427). Springer, Berlin, Heidelberg.

例えば複数のプログラム位置から呼ばれる関数があった場合、制御はそれぞれの関数呼出元から関数へ処理が移り、関数の処理が終わった後、それぞれの復帰先へと処理が戻ります。復帰後の状態から関数を呼び出す前の状態を参照するためには、呼出と復帰の経路が一致している必要があります。このように経路に解析が依存することを文脈依存性と呼びます。文脈依存性を考慮しない場合、複数ある呼出元の区別がつかないため、復帰後の状態を分析するときに無関係な文脈の情報が混入し、解析の精度が著しく低下してしまいます。文脈依存性を解決するためには、スタックを用いて関数呼出/復帰の対応の整合性を保証するなどの処理が必要になります。このような解析を「手続間プログラム解析(Inter-procedural program analysis)」と呼びます。

従って事前に関数位置が同定できていない場合においても精度の高いバイナリプログラム解析を行うためには、解析の過程において関数呼出/復帰の位置を推定する必要があります。既存の静的解析手法では、最低限の呼出規約(例えばIntel x86アーキテクチャではCALL命令を関数呼出、RET命令を関数復帰に使用しているなど)を仮定することで関数の位置を推定しています。

しかし、対象となるプログラムに前提知識がない場合、この最低限の呼出規約ですら保証することはできません。例えばCALL/RET命令を関数呼出/復帰以外に使用したり、逆にこれらの命令を他の命令に置き換えている可能性もあります。このように既存の静的解析手法をバイナリプログラム解析に適用するためには、対象となるプログラムの「素性の良さ」を保証する前提知識が必要になります。

これまでに述べてきたバイナリプログラム解析の困難さをまとめると以下のようになります。

1. 逆アセンブラでは関節ジャンプ命令の行先が分からない。
2. 関節ジャンプ命令の行先を既存の静的解析で決定するためには、事前にプログラムが関数に分割されている必要がある。
3. バイナリプログラムから関数位置を特定するには、プログラムがどのようなコンパイラを使用したか、呼出規約に従っているかなどの前提が必要になる。

こうしたことから、既存のバイナリプログラム解析は対象となるプログラムに対して作成条件などの前提知識があること、そしてそれらが信頼できることを必要としています。

本研究手法[*4]では、独自に考案した中間表現を用いた解析により、制御構造の再構成を行いながら、同時に関数と見なせるプログラム部分の特定を行うことで、対象に対する前提知識がなくても(つまり「素性の悪い」プログラムであっても)、静的プログラム解析が適用できることを目標としています。

*4 Izumida2018:Izumida, T., Mori, A., & Hashimoto, M. (2018, January). Context-Sensitive Flow Graph and Projective Single Assignment Form for Resolving Context-Dependency of Binary Code. In Proceedings of the 13th Workshop on Programming Languages and Analysis for Security (pp. 48-53).

3.3 射影的単一代入形式を用いたバイナリプログラム解析

ここでは本研究手法の概要を説明します。例として、32ビットIntel X86アーキテクチャ用の図-4のようなプログラムを用います。

[具体例]

```
00401000:   xor ecx, ecx
00401002:   mov ebx, 0x40100c
00401007:   jmp 0x401017
0040100c:   mov ebx, 0x401016
00401011:   jmp 0x401017
00401016:   hlt

00401017:   inc ecx ; (A)
00401018:   jmp ebx
```

図-4 32ビットIntel X86アーキテクチャにおける具体例

図-4の例では、関数に相当するコード(A)が2度呼び出されていますが、CALL/RET命令を用いている代わりに帰りアドレスをEBXレジスタに格納してから(A)へジャンプを行い、ECXレジスタに1を加えた後にEBXレジスタを行き先に指定してジャンプを行い呼び出し後のアドレスに復帰しています。標準的な関数呼出のようにCALL/RET命令を使用していないため、IDA Proなどの既存の解析ツールではこれらのコードが関数呼出であることを認識できません(図-5)。

本研究手法では、各機械命令を単純な代入文列に変換した後、更に「静的単一代入(Static Single Assignment、SSA)」と呼ばれる表現形式に変換します。SSAはコンパイラの最適化解

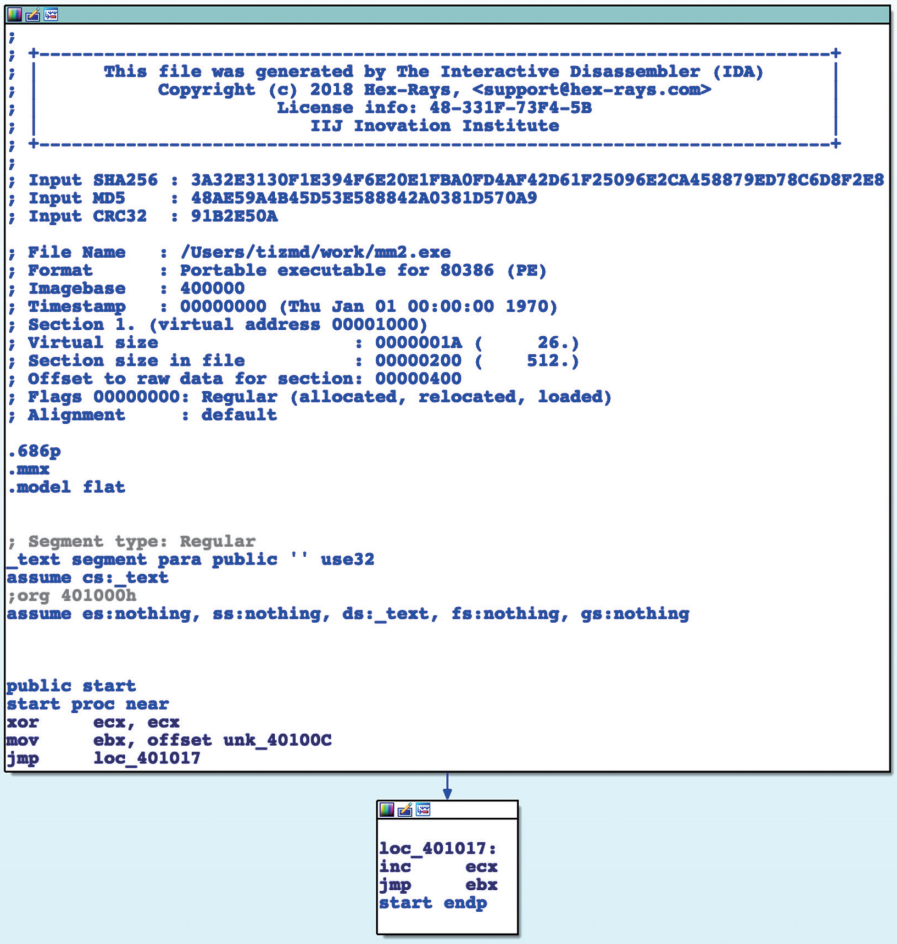

図-5 *IDA Pro*で逆アセンブルした例

析で用いられる内部表現形式で、各変数の定義が一意になるように変数名の変更を行います。これにより各変数の定義と使用(def-use)関係が明確になり、情報の流れを把握することが容易になります。例えば図-6では、2ヵ所あるECXへの代入をECX_1、ECX_4と区別しています。

ここではEBXレジスタに保存されている帰りアドレスへの間接ジャンプ命令で終了しています(ジャンプ命令はプログラムカウンタEIPへの代入として表現されています)。この場合、右辺のEBX_2の定義を遡ると0x40100cであることが容易に分かります。従って、この間接ジャンプ命令の行き先アドレスを0x40100cとして、更に展開します。

図-7は2度目の(A)の呼び出しを終えるところまでの図です。SSAではΦ関数と呼ばれる疑似関数を導入して情報の合流を表現します。例えば、$EBX_8 \leftarrow \Phi_8(3{:}EBX_2, 7{:}EBX_6)$という代入文ではノード3から来たEBXレジスタの値EBX_3とノード7から来た値EBX_6が合流して新たにEBX_8という変数に代入されています。先ほど同様にEBX_2の定義を遡ると、EBX_2はΦ関数を用いて、$\Phi_8(3{:}0x40100c, 7{:}0x401016)$として表現されます。これはすなわち、制御がノード8にノード3から来た場合のEBXレジスタの値は0x40100c、制御がノード7から来た場合は0x401016と情報の合流があることを表しています。このように特定の時点で合流した情報によって行き先アドレスが変わる場合を、本研究では「文脈依存性」として定義します。この場合、0x401017から0x401018の範囲のコードは、複数の文脈から再利用されているので、ここが「関数」であると推定することができます。

こうして文脈依存性が検出できた場合、本手法では更にΠ関数と呼ばれる疑似関数を挿入します(図-8)。Π関数はΦ関数に対する射影関数として働きます。例えば$\Pi_{3\to8}(\cdots)$という式

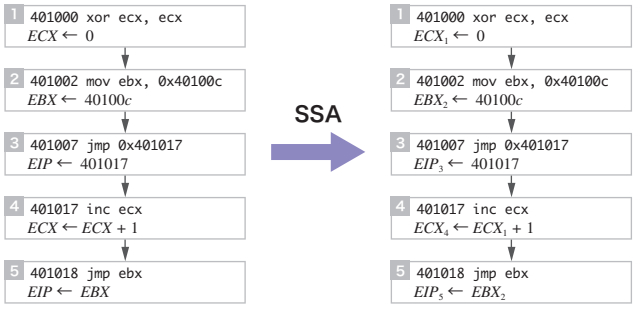

図-6 SSA形式:最初の(A)の呼出を終えるところまで

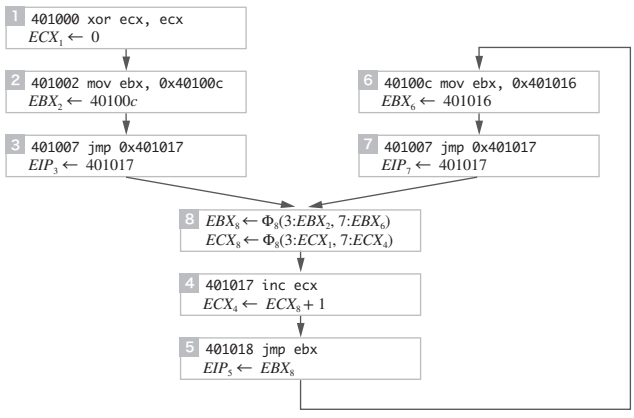

図-7 SSA形式: 2度目の(A)の呼出を終えるところまで

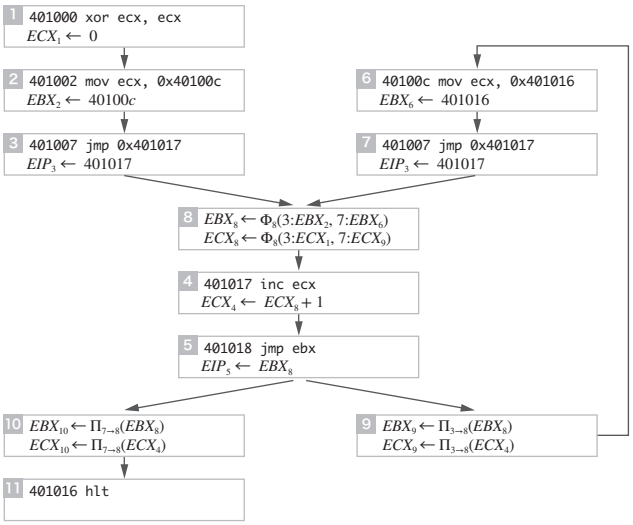

図-8 PSA形式:Π関数の挿入

はノード8で合流した情報からノード3から来た情報を取り出す、という意味をもち、$\Pi_{3\to8}(\Phi_8(3{:}X,7{:}Y))\Rightarrow X$のように評価されます。このΠ射影関数によるSSAの拡張は「射影的単一代入(Projective Single Assignment、PSA)」と呼ぶ本研究独自の表現形式です。

Π関数を利用すると、例えばプログラム終了時(ノード11)のECXレジスタの値は以下のように計算できます。

$$ECX_{10}\Rightarrow \Pi_{7\to8}(ECX_4)\Rightarrow \Pi_{7\to8}(ECX_8)+1\Rightarrow \Pi_{7\to8}(\Phi_8(3{:}ECX_1,\,7{:}ECX_9))+1$$
$$\Rightarrow ECX_9+1\Rightarrow \Pi_{3\to8}(ECX_4)+1\Rightarrow \Pi_{3\to8}(ECX_8)+2\Rightarrow \Pi_{3\to8}(\Phi_8(3{:}ECX_1,\,7{:}ECX_9))+2$$
$$\Rightarrow ECX_1+2\Rightarrow 2$$

このように本研究手法では呼出規則に従っていないようなプログラムでも、再構成の過程で文脈依存性を検出し、解決することが可能となっています。

3.4 応用:バッファオーバーフローの安全性証明

射影的単一代入では射影関数Πだけでなく、各条件分岐における分岐条件を値に付与する条件関数Γも追加されます。これを応用すると、関数内でループによってスタック上のデータを書き換えるような処理を行っている場合、ループの中でスタック上に記録されている帰りアドレスが上書きされる可能性があるかどうかを調べることができます。

例えば、図-9のプログラムをPSA形式に変換して簡略化すると図-10のようになります。

ここでLd(M,A,N)はメモリ状態Mのアドレスからのnバイトの値の読み出しを、St(M,A,X,N)はメモリ状態Mのアドレスへのnバイトの値Xの書き込みを表現しています。ここでは関数の終了時にスタックメモリ上のEBP_1+4と表現された位置に格納された4バイトの復帰アドレス値へジャンプしています。この領域がforループの中で上書きされる条件は、PSA形式を用いて以下のように表現されます。

$$\Gamma(i_2<10,\ EBP_1+4\le EBP_1-10+i_2\le EBP_1+8)$$

これは$i_2<10$の条件のもとで$EBP_1+4\le EBP_1-10+i_2\le EBP_1+8$が成立するか、ということを表します。$i_2$はΦ($i_1$,$i_4$)と定義されており、$i_4$がループ内で定義されているため、$i_2$はループ内で変動する値であることが表現されています。この条件を満たすi_2の値があれば、ループ中に復帰アドレスの値が上書きされる可能性があります。しかし、この場合は容易にそのようなi_2の値が存在しないことが分かります。つまり、このループによって復帰アドレスの領域が変更されることはないことが保証されます。

実際のプログラムではループ条件もメモリ書換条件もより複雑な形で現れます。ループ内で変動するこれらの条件が充足可能か否かを決定するためには、ループ不変条件(Loop invariant)と呼ばれるループの中では変わらない条件式を見つけることが重要になります。近年こうしたループ不変条件を自動的に算出する解析手法の研究が進み、Z3などSMTソルバ内に実装されています。本研究ではバイナリプログラムからループ条件や

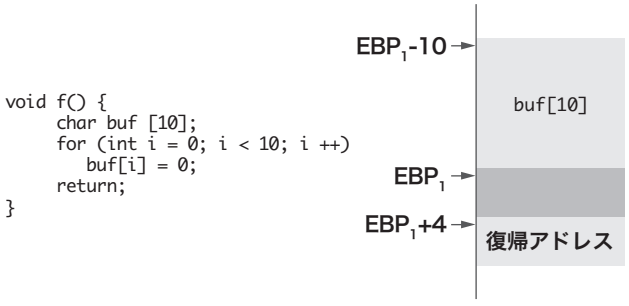

図-9 プログラム例

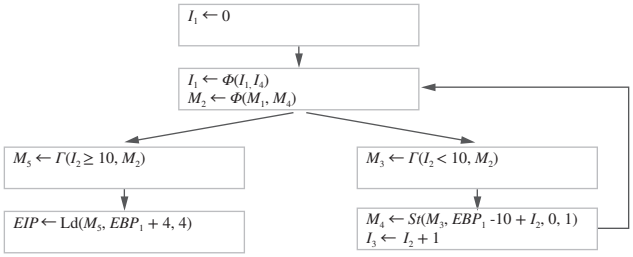

図-10 ループ

メモリ書換条件を抽出し、SMTソルバを用いてループ不変条件を自動算出することを目標としています。このような解析は必ずしも一定時間内に決定できるとは限りませんが、もし上書きされる可能性がないと判定できた場合は、バッファオーバーフローが起こらないということを保証できることになります。また、上書きされる可能性があると判定した場合はどのような条件下で上書きが起こり得るのかを調べることができます。

現在本手法を用いてAIエッジデバイスのような組込ファームウェアに対し、バッファオーバーフローのなどの脆弱性やトロイの木馬のようなバックドアの有無を検出する技術への応用を研究しています。

3.5 おわりに

本稿では技術研究所で開発を行っている、前提知識を必要としないバイナリプログラム解析技術について解説しました。既存の静的解析ではプログラムを事前に関数単位で分割する必要がありますが、そのためにはプログラムがどのように生成されたかについて前提知識を必要としました。本研究手法では、独自拡張されたSSA形式を用いて間接ジャンプの行き先を評価すると同時に文脈依存性を検出することで関数領域の推定を行うことができるため、前提知識が得られない「素性の悪い」プログラムであってもプログラム解析を行うことが可能となっています。

しかし、静的バイナリプログラム解析には決定不能問題が含まれているため、完全解が得られる解析手法は存在しません。本手法においても、評価式が肥大化し静的な解決が困難になりそうな場合にはそれ以上の評価を行わないことで、近似解を出しています。

バイナリプログラム解析に用いられる手法としては他に記号実行(Symbolic execution)が挙げられます。記号実行は静的解析と動的解析の中間に位置する解析手法で、2016年にDARPAが主宰した情報セキュリティの自動化コンテストCyber Grand Challengeでは、mayhem[*5]やangr[*6]と行った記号実行を用いた解析ツールが上位に入りました。記号実行ではプログラムがバッファオーバーフローなど危険な状態になりうるかどうかを検出することができます。しかし、プログラムが安全である、すなわち危険な状態にはなり得ないことを証明するには網羅的な実行が必要になるため記号実行には向いていません。その点で記号実行と本研究手法は補完的な役割を果たすと考えられます。

今後は本研究と記号実行や動的解析など他の解析手法を統合したバイナリプログラム解析ツールの開発を目指していきます。

《 謝辞 》
この研究は国立研究法人新エネルギー・産業技術総合開発機構(NEDO)「AIエッジデバイスの横断的なセキュリティ評価に必要な基盤技術の研究開発」の委託業務の一部として行っています。

執筆者:
泉田 大宗(いずみだ とものり)
IIJ-II技術研究所 研究所員(2015年より)博士(情報科学)。

*5 Cha2012:Cha, S. K., Avgerinos, T., Rebert, A., & Brumley, D. (2012, May). Unleashing mayhem on binary code. In 2012 IEEE Symposium on Security and Privacy (pp. 380-394). IEEE.
*6 Wang2017:Wang, F., & Shoshitaishvili, Y. (2017, September). Angr-the next generation of binary analysis. In 2017 IEEE Cybersecurity Development (SecDev) (pp. 8-9). IEEE.