AdaCore TECH PAPER

Ada、SPARK、Rust言語のメモリ安全性

Ada、SPARK、Rust言語のメモリ安全性

概要

メモリ・セーフティに関わるバグは、ソフトウェアのセキュリティや信頼性に深刻な影響を与える原因となります。実際、Google や Microsoft などの大手テクノロジー企業は、報告されているセキュリティ脆弱性の70%以上が、こうした問題に関連しているとしています。これらのリスクを効果的に減らすためには、プログラミング言語の設計段階でメモリ・セーフティを確保することが非常に重要です。

本ホワイトペーパーでは、境界外書き込みやユーズ・アフター・フリー、ヌル・ポインタ参照といった、よく見られるメモリ・セーフティのバグについて説明します。あわせて、Ada、SPARK、Rust といったプログラミング言語が、これらの問題にどのように対応しているのかを解説します。

Ada と SPARK は、強力な型システムを持つプログラミング言語で、言語機能や実行時の検証、静的解析ツールを活用して、包括的なメモリ・セーフティを実現します。Ada は、配列の境界チェックや安全なポインタの仕組みによって、安全なメモリ管理をサポートします。一方、SPARK は静的解析や形式検証を用いて、メモリ・セーフティに関わるリスクの回避を図っています。

Rust は、所有権モデルや借用ルール、そして強力なコンパイル時の検証機能によって、メモリ・セーフティに対する新しいアプローチを提供しています。Rust では、配列の境界チェックやヌル・ポインタを原則として排除する設計により、安全なメモリの扱いが可能になります。これらの仕組みは、セキュリティと高いパフォーマンスの両方が求められるシステムにおいて、特に重要な役割を果たします。

Ada、SPARK、Rust のようなメモリ・セーフティを重視したプログラミング言語を採用することで、メモリに関連するバグを減らして、より安全で信頼性の高いソフトウェアシステムを構築することが可能になります。これらの言語は、ソフトウェア開発における脆弱性の回避が可能で、複雑なシステム開発に貢献します。

はじめに

メモリ・セーフティに関するバグは、長年にわたり、サイバー・システムのセキュリティや安全性に深刻な脅威を与えてきました。プログラミング言語はこの問題において重要な役割を果たしており、場合によっては問題の原因となることもあれば、有効な解決策となることもあります。Google（2021年）や Microsoft（2019年）は、セキュリティバグの70%以上がメモリ・セーフティに関連していると報告しています。

メモリ・セーフティは、ソフトウェア開発における重要な要素であり、アプリケーションがメモリ操作（リード、ライト、アロケーション、デアロケーション）をどのように処理するかに関係しています。メモリ・セーフなアプリケーションは、割り当てられたメモリの範囲内で動作し、許可されていないメモリ領域にアクセスしたり書き換えたりせず、データ競合を回避し、使用後にメモリを適切に解放します。メモリの管理が不適切であると、アプリケーションのクラッシュや、攻撃者が悪用できるセキュリティ脆弱性を引き起こす可能性があります。

多くのソフトウェア・バグは、アプリケーションに割り当てられたメモリ空間外での読み書きを引き起こす可能性があります。このようなバグを回避するためには、手動での対策か、メモリ・セーフなプログラミング言語を使用して安全性を確保する必要があります。しかし、手作業で対策を行うのは手間がかかり、エラーが発生しやすくなります。そのため、動的および静的な組み込みチェックやツールを活用し、手作業を最小限に抑えつつ、目標を達成する方法が推奨されます。これは、米国家安全保障局（NSA）の勧告にも従ったアプローチです。

メモリ・バグとは

境界外書き込み:

境界外書き込みは、プログラムに割り当てられたメモリ領域の範囲外にデータを書き込むことで発生し、隣接するメモリ領域を破壊する可能性があります。Common Weakness Enumeration 2023によれば、これは非常に危険なソフトウェアの脆弱性の一つです。この書き込みが、他のサブプログラムで使用されるデータや関数のリターン・アドレスといったセンシティブな情報に対して行われることがあります。攻撃者は、リターン・アドレスを上書きすることでプログラムの実行フローを操作し、悪意のあるコードを実行することができます。このバグの根本的な原因には、管理されていないポインタ演算やダングリング・ポインタなどの問題が関与している場合があります。

ユーズ・アフター・フリー （ダングリング・ポインタ参照（デリファレンス））:

「ユーズ・アフター・フリー」バグは、プログラムがすでに解放されたメモリにアクセスすることで発生します。すでに解放されたメモリ領域を指しているポインタは「ダングリング・ポインタ」と呼ばれ、これを参照するとさまざまな問題を引き起こす可能性があります。もしそのメモリがすでに他の目的で利用されていた場合、ダングリング・ポインタを使ってアクセスすると、正しく動作しているデータが破壊されてしまう（上書きされる）ことがあります。さらに、その新しいデータが関数の戻りアドレスであった場合、悪意あるコードを実行させる攻撃に悪用される危険性もあります。たとえ書き込みをしなくても、再利用されたメモリを読み取るだけで機密情報が漏洩する可能性もあります。

境界外読み取り:

境界外読み取りは、アプリケーションが本来のバッファ範囲を超えてデータを読み込むことで発生します。このような操作は、プログラムの予期しない動作を引き起こし、セキュリティ上の脆弱性を生む原因となります。特に重大な問題は、機密データへの不正アクセスです。さらに深刻な脅威として、攻撃者が任意のコードを実行し、アプリケーションをクラッシュさせる可能性があります。

ヌル・ポインタ参照（デリファレンス）:

ヌル・ポインタ参照は、プログラムをクラッシュさせ、予期しない動作を引き起こす原因となることがあります。アプリケーションが有効なメモリ・アドレスを指していないポインタを使用してデータにアクセスし、適切な例外処理が行われない場合、通常はクラッシュが発生します。

整数オーバーフローまたはラップアラウンド:

整数の算術演算の結果、型の範囲を超える値が生じた場合、その値は演算に応じて最大値または最小値にラップアラウンドされます。このような計算結果がリソース管理に使用されると、バッファ・オーバーフローや無限ループ（値がループインデックスとして使用される場合）、さらにはメモリやCPUといったリソースの枯渇を引き起こす可能性があります。

データ競合:

競合状態は、排他的にアクセスされるべきリソースが、並行して実行される複数のスレッドによって同時に変更または読み取られる状況を指します。この問題は、スレッド間の同期が欠如していることが原因で発生し、共有リソースの予期しない状態やアプリケーションのクラッシュ、さらにはリソースの枯渇を引き起こす可能性があります。

メモリ・リーク:

メモリ・リークは、コンピュータ・プログラムがメモリを誤って割り当てることで発生します。具体的には、プログラムがメモリを割り当てた後、不要になったメモリを解放しない場合に起こります。時間が経過するにつれて、アプリケーションのメモリ消費量が増加し、パフォーマンスの低下や、極端な場合にはシステム・クラッシュを引き起こすことがあります。メモリ・リークは、CやC++のように手作業でメモリ管理を行う言語で特に発生しやすい問題です。これらの言語では、プログラマがメモリの割り当てと解放を明示的に行う必要があり、メモリを誤って解放しないことで、不要なメモリを使用する状況が発生することがあります。対照的に、AdaやRustのような言語では、より安全なメモリ管理が行われています。

メモリ・セーフ・プログラミング言語

メモリ・セーフ・プログラミング言語は、コードを作成する際に手作業でメモリの誤使用を確認するのではなく、誤使用を避けるための仕組みを提供しています。これには、境界検証のような基本的な手法から、変数の所有権管理のような高度な手法までさまざまなものがあります。さらに、これらの言語は、バグの発生を防ぎ、プログラマが手動で確認することなく、メモリを自動的に管理します。こうして、メモリ・セーフに関する問題を解決できますが、各言語における保護のレベルやメカニズム、サポートツールには違いがあります。

Ada と SPARK は、言語機能、実行時検証、静的および動的解析ツールのサポートを組み合わせることで、メモリの安全性を実現する強力な型付け言語です。
Ada は、安全性が重要なアプリケーションの開発に広く使用され、安全性に特化したプログラミング言語です。この言語は、実行時およびコンパイル時の検証機能を備えた言語構造により、メモリの安全性を確保します。強力な型付け、形式パラメータやモード、保護されたオブジェクト、安全なポインタなどが、Ada の主要な機能です。

Ada のビルドイン検証機能により、バッファ・オーバーフローや境界外読み書きなどのエラーを回避することができます。強力な型付けとパラメータ・モードは、メモリの誤った変更や制御フローの変更を防ぎ、不正なデータの混入を回避します。保護されたオブジェクトは、共有リソースへのアクセスが安全であり、競合状態が発生しないことを確実にします。
他の言語では多くのメモリ・バグを引き起こす原因となるポインタですが、Ada では安全なアクセス・ルールとヌル・ポインタの参照（デリファレンス）を回避するヌル値除外機能を使用して、ポインタをより安全に扱うことができます。さらに、ポインタを使用せずに、同様の機能を提供しながら、オーバーヘッドや潜在的なメモリ安全性の問題を解消する手法もあります。

さらに、Adaのメモリ安全性は、実行時やコンパイル時に特定するのが困難なエラーを検証する動的・静的解析ツールによって裏付けられています。SPARKはAdaのサブセットで、米国国立標準技術研究所（NIST）で、メモリ安全言語として登録されています。 その違いは、Adaでは多くの検証が実行時に行われるのに対し、SPARKではすべての検証が静的に行われます。SPARKには、所有権メカニズムに基づく安全なポインタ機能があります。

Rust は、変数の不変性、所有権、借用、境界検証などの仕組みを通じて、メモリの安全性を実現しています。
Rust では、変数はデフォルトで不変（変更できない）であるため、意図しない変更を防止します。
Rust コンパイラーは、プログラム内に存在する各値に対して唯一の「所有者」が存在できる所有権システムを採用し、効率的かつ安全なメモリ管理を実現しています。所有者がスコープ（有効範囲）を離れると、そのメモリは自動的に解放されます。また、Rust の借用の仕組みにより、複数の参照から変数にアクセスすることができますが、「同時に変更できる参照は1つだけ」というルールがあるため、並行処理（スレッド）においても競合状態を防ぐことができます。
Rust では、コンパイル時に各ポインタが有効なメモリを参照しているかどうかがチェックされます。つまり、ポインタが何かしらの値を正しく指していることが確認され、無効なポインタ参照によるエラーを回避できます。
Rust にはヌル・ポインタという概念がなく、ヌル・ポインタを参照（デリファレンス）することもできません。さらに、Rust では実行時に配列やベクタへのアクセスが範囲内で行われているかどうかを確認する「境界検証」が自動で行われます。これにより、バッファ・オーバーフローといったエラーを回避できます。
また、Rust には「unsafe Rust」と呼ばれるモードがあり、特別なケースでより柔軟な操作が可能になります。ただし、このモードを使っても、借用チェックや境界チェックといった基本的な安全機能は維持されます。

Ada言語におけるメモリ・バグへの対応

Ada は、一般的なメモリ・バグを軽減するために、多くの堅牢な安全機能を備えています。以下に、Ada がこれらの問題にどのように対処しているかを示します。

1. 境界外書き込み

Ada では、配列の境界外アクセスを防ぐために、実行時に境界チェックが行われます。プログラムが配列の有効範囲外にアクセスすると、Ada は Constraint_Error という例外を発生させて、処理を中断します。この仕組みによって、メモリ破壊やセキュリティ上の深刻な脆弱性を回避できます。

2. ユーズ・アフター・フリー

Ada では、ほとんどの場面でプログラマが手動でメモリを管理する必要がありません。代わりに、制御された型やストレージプールといった仕組みにより、動的メモリをより安全に扱うことができます。また、強力な型付けとスコープに基づいたメモリ管理により、解放されたメモリを誤って再利用してしまうようなバグ（ユーズ・アフター・フリー）の発生リスクを大きく減らしています。

3. 境界外読み取り

Ada では、配列や文字列などのデータにアクセスする際、境界内にあるかを実行時にチェックします。もし境界外にアクセスすると、例外を発生し、処理を中断します。これにより、未定義の動作や安全でない状態に陥るのを回避できます。

インデックス・チェック:

Ada は実行時に、配列アクセスの境界チェックを行い、使用されるインデックスが配列の範囲内であることを確認します。これにより、境界外のメモリへのアクセスを防ぎ、バッファ・オーバーフローを回避します。もしインデックスが不正であれば、Constraint_Error が発生し、未定義や安全でない動作を避けることができます。

インデックス・チェックが失敗する例

```
procedure  Ex_Index_Check is
      type Integer_Array is
          array (Positive range <>) of Integer;
      function Inc_Value_Of (A : Integer_Array; I : Integer)
                                          return Integer
      is
            begin
            return A (I) + 10;
      end Inc_Value_Of;
      Arr_1 : Integer_Array (1 .. 10) := (others => 1); Arr_2 : Integer_Array (1 .. 5) :=
      (others => 0);
begin
      -- raises Constraint_Error as Index Check fails
      -- in the call of Inc_Value_Of (Arr_2, 10) Arr_1 (10) := Inc_Value_Of
      (Arr_2, 10);
end Ex_Index_Check;
```

オーバーフロー チェック:

Ada はスカラーオブジェクトの値が型の指定された範囲内であることを確認します。ラップアラウンドは行わず、意図しないメモリ問題やバグを回避します。もし値が範囲外の場合、Constraint_Error が発生します。ラップが必要な場合は、モジュラー（符号なし）整数型を使用できます。

オーバーフロー チェックが失敗する例

```
procedure  Ex_Overflow_Check is A :
   Integer := Integer'Last;
begin
      -- raises Constraint_Error due to integer wraparound A := A + 1;
end   Ex_Overflow_Check;
```

範囲チェック:

Ada では、ユーザ定義の型やサブタイプに対して、特定の範囲（型やサブタイプの値に対する不変条件）が設定できます。これは、サイズ、長さ、周波数、速度など、アプリケーション固有の範囲を持つ値を表現する際に有効です。範囲チェックは、変数への値の割り当てがその型またはサブタイプの不変条件に違反していないことを確認します。もし範囲外の値が割り当てられた場合、未定義の動作が発生する可能性があり、安全でないメモリアクセスが発生するリスクがあります。

範囲チェックが失敗する例

```
procedure Ex_Range_Check is
      subtype Sub_Int is Integer range 1 .. 50; I : Sub_Int;
begin
      I := 52;   -- raises Constraint_Error end Ex_Range_Check;
```

ストレージ チェック:

新しいオブジェクトがスタックやストレージプールに作成される際、十分なメモリ領域が確保されているかを確認します。もし必要なメモリ領域が不足している場合、Storage_Error 例外が発生します。このチェックは、境界外の読み取りや書き込みを防止し、メモリの安全性を確保するために重要な役割を果たします。

判別チェック（Discriminant Check）:

バリアントレコードのフィールドへアクセスする際の安全性を確保します。プログラムが、バリアントレコードの有効でないフィールドにアクセスすると Constraint Errorが発生して、Reliance on Data/Memory Layout や Access of Resource Using Incompatible Type (Type Confusion) エラーを回避します。

判別チェック（Discriminant Check）が失敗する例

```
procedure Ex_Discriminant_Check is
      type Message_Type is (Number, Pointer);
           type Message_Buffer (MT : Message_Type) is record case MT is
        when Number =>
                     Num : Integer; when
        Pointer =>
                     Ptr : access Integer;
end case;
      end record;
        Number_MB : Message_Buffer (MT => Number); Var : Integer :=
      0;
begin
      Number_MB.Num := 21; -- OK
      Var := Number_MB.Ptr.all;   -- raises Constraint_Error end
Ex_Discriminant_Check;
```

長さのチェック（Length Check）:

配列の割り当てにおいて、両方の配列が同じ長さであることを確認します。Ada では、配列間の代入や比較において、両方の配列の長さが一致していることを実行時に検証します。このチェックにより、境界外の読み書きが防止され、安全な配列操作が確実となります。

長さのチェックに失敗する例

```
procedure Ex_Length_Check is type
      Integer_Array is
          array (Positive range <>) of Integer;

            Arr_1 : Integer_Array (1 .. 10); Arr_2 :
             Integer_Array (1 .. 9) := (others => 1);
begin
    -- Length Check fails --> raises Constraint_Error Arr_2 := Arr1;
    end Ex_Length_Check;
```

アクセシビリティ・チェック:

このチェックは、宣言されたオブジェクトを指すアクセス値に対するダングリング参照を防止するものです（このチェックはサブプログラムを指すアクセス型にも適用されますが、それはメモリ管理には関係しません）。アクセス・オブジェクトは対応する型よりも長く存在できないため、このチェックは一般的にコンパイル時に行うことができます。コンパイラーはその結果を使用して、ダングリング参照の可能性があるかどうかを判断できます。

4. ヌル・ポインタ参照（デリファレンス）

Ada は、ポインタを扱う際にヌル値（null）のチェックを行うことで、このリスクを軽減しています。Ada の アクセス型（ポインタ）は、ヌル値チェックを標準で備えており、ヌルのアクセス値を参照すると Constraint_Error例外が発生します。そのため、ヌル・ポインタを参照する可能性は低くなります。他の多くの言語では、ポインタは多くのメモリ・バグの主な原因となっています。たとえば、ヌル・ポインタや範囲外のポインタの参照、解放後のメモリの使用、二重解放などが挙げられます。

Adaは、以下の方法でポインタの危険性を回避します：

Ada のポインタは、「アクセス型」と呼ばれる仕組みで、特定のコンピュータに依存しないように設計されています。アクセス型の値は、初期状態では何も指していませんが、動的に確保されたオブジェクト（ヒープ上のオブジェクト）や、プログラム内で宣言されたオブジェクト（スタック上のオブジェクト）を指すことができます。動的に確保されたオブジェクトを指す場合は、他の多くの言語と同じように new という予約語を使ってアクセス値を作ります。一方、宣言されたオブジェクトを指したいときは、そのオブジェクトに Access 属性を使用して、アクセス値を作ることができます。

Ada では、アクセス値を作る方法が定められており、特別な（安全でない）型変換を使わない限り、自由にポインタを作ることはできません。そのため、C 言語などのようにアドレスを直接操作するよりも、安全です。また、アクセス値に対応するメモリ解放を行う場合、安全チェックを行わない特殊な機能を使用する必要があります。そのため、アクセス値には意味があり、無効なポインタ参照が回避されます。また、ポインタの代替手段を提供しています。例えば、パラメータ・モード、制約のない配列、ヒープを使用しない動的サイズのオブジェクトなどは、典型的なポインタの使い方を置き換えることができます。

5. 整数オーバーフロー

Adaは整数のオーバーフローに対して、コンパイル時と実行時の両方のチェックを行います。デフォルトでは、特定のプラグマで明示的にチェックを無効にしない限り、 Adaはオーバーフローが発生した時に Constraint_Error 例外を発生します。

6. データ競合

Adaは、プロテクトオブジェクト（protected objects）とランデブ（rendezvous）というタスクモデルに基づき、データ競合を回避します。プロテクトオブジェクトは相互排他を強制することで、共有データへの安全な同時アクセスを提供します。さらに、Adaの強力な型付けとタスク構造により、競合状態の発生を軽減し、安全な並行プログラミングを実現します。

7. メモリ・リーク

Ada は、プログラムの中で動的にメモリを確保しなくてもすむように設計された言語です。高機能で抽象的な型(abstract)が用意されているため、動的にオブジェクトを作る必要がありません。たとえば、変数のサイズが実行時に決まるような場合でもスタック上で扱うことができ、サイズが決まっていない型も、関数の戻り値として返すことができます。また、配列やリストといったコレクション型では、メモリの管理を自動処理します。

動的メモリ割り当てを行わないように、プログラムで制限することも可能です。動的メモリが必要な場合でも、Ada はスコープから外れた際に自動で処理（クリーンアップ）が可能な「制御型（controlled types）」を用意しています。また、ストレージプールと呼ばれる機能で、動的に割り当てられたオブジェクトをまとめて解放することができます。

これらの安全機能を取り入れることで、Adaでは一般的なメモリ・バグのリスクを大幅に低減し、信頼性が高い安全なソフトウェアシステムの開発を行えます。

Rust言語におけるメモリ・バグへの対応

Rustは、メモリ安全性を保証する言語として設計されています。以下に、Rustがメモリ安全性を実現するために採用している主な仕組みを示します。

1. 境界外書き込み

Rustの安全機構、特に所有権モデルと境界チェック付きのインデックス処理は、境界外書き込みを効果的に回避します。配列やスライスの範囲を超えてアクセスすると、Rustは実行時にパニックを発生させて処理を停止し、メモリ破壊を回避します。

2. ユーズ・アフター・フリー

Rustの所有権と借用システムは、メモリがスコープ外になると自動的かつ安全に解放します。Rustコンパイラーは、所有権、借用、およびライフタイムに関して厳格なルールを適用しており、これにより「解放後の使用」エラーを効果的に回避します。メモリは、アクセスできなくなったときにのみ解放されるため、解放されたメモリを使用することは不可能です。

3. 境界外読み取り

Rustの配列とスライスの組み込み境界チェックは、読み取り操作が有効なメモリ範囲内にあることを確実にします。範囲外の読み取りが実行されると、Rustはパニックを引き起こし、意図しないデータアクセスや未定義の動作を回避します。

8. ヌル・ポインタ参照（デリファレンス）

ヌル・ポインタ参照（デリファレンス）は、プログラムがヌル・ポインタを通じてメモリ位置にアクセスする際に発生し、クラッシュや予期しない動作を引き起こします。Rustは、ヌル・ポインタを許可しないことで、このリスクを排除します。代わりに、RustはOption<T>型を使用して、値が存在しない可能性を表現します。この型は、値が存在しない可能性がある場合に、プログラマに明示的に処理させることで、NULLポインタの参照（デリファレンス）を回避します。

4. 整数オーバーフロー

Rustは整数オーバーフローを処理するための2つのモードを提供しています。デバッグビルドでは、Rustは算術演算中にオーバーフローをチェックし、オーバーフローが起きるとパニックが発生します。リリースビルドでは、Rustはデフォルトでこのチェックを行いませんが、オーバーフローを明示的に処理できるように、チェック、ラッピング、飽和算術演算を提供しており、整数演算は安全で予測可能です。

5. データ競合

データ競合は、2つ以上のスレッドが共有データに同時にアクセスし、そのうち少なくとも1つのアクセスが書き込みで適切な同期が行われていない場合に発生します。Rustの所有権モデルと型システムは、コンパイル時にデータ競合を回避します。Rustは厳格な借用ルールを適用し、データが安全ではない方法で複数のスレッドからアクセスされるのを回避します。さらに、RustはMutexやRwLockなどの並行性プリミティブを提供しており、これらは共有データへの安全なアクセスを保証します。コンパイラーはこれらのプリミティブが正しく使用されているかをチェックし、データ競合の可能性を排除します。

6. メモリ・リーク

Rustの所有権モデルでは、メモリがスコープ外になると自動的に解放され、厳格なメモリ安全性を確保します。
Rustはコンパイル時に厳格なルールを適用して、他のプログラミング言語で一般的に見られる多くのメモリ・バグを回避できるため、信頼性の高い安全なシステムの開発に最適な選択肢となります。

SPARK言語におけるメモリ・バグへの対応

SPARK（形式検証言語）は、Adaのサブセットで、安全性とセキュリティが重要な高信頼性システム向けに設計されています。SPARKは、制限的なプログラミング・モデル、静的解析、および形式検証を通じて、多くの一般的なメモリ・バグを排除します。以下に、SPARKがメモリの問題にどのように対処しているかを示します：

1. 境界外書き込み

SPARKは、コンパイル時に静的解析を行うことで、このリスクを排除します。形式検証を用いて、SPARKは配列アクセスが有効な範囲内であることを確認します。範囲外への書き込みが試行されると、プログラム実行前に検出され、検証されたSPARKコードでは、このエラーが発生することはありません。

2. ユーズ・アフター・フリー

SPARKは、動的メモリの割り当てや解放をサポートしないことで、この問題を回避します。メモリの誤った管理につながる操作を禁止することで、SPARKはメモリが予測可能で安全な方法で割り当ておよび使用が可能で、解放後の使用エラーを効果的に防止します。

3. 境界外読み取り

SPARKでは、配列の境界は形式手法を使用してコンパイル時に厳密にチェックされます。この静的解析により、すべての読み取りがデータ構造の有効範囲内で行われるため、SPARKコードでの境界外読み取りの可能性が排除されます。

4. ヌル・ポインタ参照（デリファレンス）

SPARKでは、ヌル・ポインタ参照（プログラムがヌル・ポインタを通じてメモリにアクセスする状況）は排除されています。言語仕様でヌル・ポインタの使用が禁止されており、ポインタに相当する構造（Adaでの アクセス型）は適切に初期化され、検証されることが求められます。そのため、すべての参照（デリファレンス）操作が有効です。

5. 整数オーバーフロー

SPARKは、整数演算が安全な範囲内で実行されることを静的解析によって検証して、整数オーバーフローを回避します。変数の期待される範囲を明示的に指定する必要があり、SPARKの解析ツールはその範囲がプログラム全体で遵守されているかを検証します。この形式検証により、SPARKに準拠したコードでは整数オーバーフローが発生しません。

6. データ競合

SPARKは、制限された並行性モデルによってデータ競合の可能性を排除します。SPARKは、並行プログラムにおけるデータアクセスに対して厳格なルールを適用し、形式検証によってそのルールが順守されていることを確認します。共有データへの非同期アクセスを許可しない設計により、SPARKはデータ競合を回避します。

7. メモリ・リーク

SPARKは、メモリ誤用につながる可能性のある特定のプログラミング構文を除外することで、メモリ安全性をさらに強化します。形式検証を活用し、安全でないプログラミング手法を除外することで、SPARKは一般的なメモリ・バグの発生を低減するのではなく、検証済みコードで発生することを回避します。この特性により、SPARKは信頼性とセキュリティが優先されるアプリケーションに最適な選択肢となります。

まとめ

結論として、メモリ安全性は、特にメモリに関連するバグが致命的な結果を引き起こす可能性のある環境において、安全で信頼性の高いソフトウェアシステムを開発するために不可欠です。

Ada、SPARK、およびRustは、強力な型付け、コンパイル時の検証、ならびに実行時の保護機能を通じて、最新のプログラミング言語が一般的なメモリ安全性の問題を軽減、あるいは排除できることを実証しています。

Ada は、豊富な実行時チェックと安全な言語構文を備えており、メモリの管理を効果的かつ安全に行うことができます。Ada のサブセットである SPARK は、静的解析と形式検証を用いて、メモリ安全性のリスクをプログラム実行前に排除するため、高信頼性が求められるシステムに最適です。

一方、Rust は革新的な所有権モデルと借用ルール、安全機構を備えており、安全性と性能のバランスを実現しています。これにより、セキュリティと効率性の両方が求められるシステム開発において、理想的な選択肢となります。

メモリ安全性が重要な課題である中、Ada、SPARK、Rust のような言語を採用することは、メモリ関連のバグの発生率を大幅に減らし、より安全で信頼性の高いソフトウェアの実現につながります。

これらの言語は、メモリ安全を言語レベルで実現しており、開発者は発見しにくい、深刻な脆弱性から保護されているという安心感のもと、システム開発に集中することができます。

*本資料は、AdaCoreの資料を意訳したものです。正確な内容については、原文こちらをご参照下さい。

AdaCore

adacore.com

IT Access®
アイティアクセス株式会社

本社　〒222-0033　神奈川県横浜市港北区新横浜3-17-6
TEL:045-474-9095 FAX:045-474-8823
URL: https://www.itaccess.co.jp

記載の会社名、製品名は、各社の登録商標または商標です。

202504