SecHack365

開発駆動コース
川合ゼミ
12Dk 奥野凌汰

ゲーム開発者がチート対策を意識しなくていい言語

Naughtiness(ナギ)

※Naughtinessはいたずらという意味

ナギちゃん(Illustrated by 私)

1. 概要

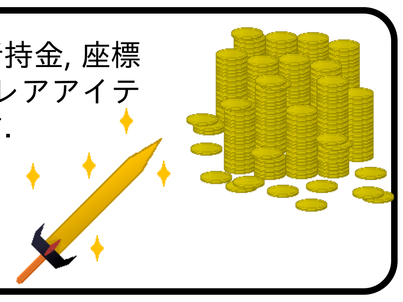
このポスターを見ている皆さんはゲームを作成したことがありますか? もしゲームを作成したことがなかったら是非作りましょう. ゲームを作成したことがある方は「どうギミックを作るか」「どうすれば面白くなるか」「〇〇の部分にこだわりたい」など, ゲーム作りのことばかり考えていませんでしたか? そんな時にチート対策を組み込んでくれと言われたらやりたいですか? 「チート対策を考える時間があるなら, 少しでもゲームを面白くすることに時間を使いたい!!!」と考えませんか? そこで自作言語の**Naughtiness(ナギ)**では, **クライアントサイドのチート対策を開発者の見えない部分で自動的に組み込む**ため, ゲーム開発者はゲーム作りに専念するだけで, チート対策済みのゲームを作成できます.

チートとは? (ざっくり)

チートは「騙す」「欺く」といった意味がありますが, コンピューターゲームでは不正な改造を施し, 自分のプレイが有利になるような改変を行う行為の意味で使用される場合が多いです.

2. チート行為の例

CheatEngineなどのツールを用いてプレイヤーのステータスや所持金, 座標などをゲーム開発者の意図しない値に書き換えることで, 不正にレアアイテムを入手したり通常ではあり得ない攻撃力にすることができます.



ほかにも以下のような例があります
- セーブデータの書き換え
- 実行中のメモリ書き換え
- バイナリファイルの書き換え

など

チート行為の対策例

※今後のバージョンで対応予定の内容です

セーブデータの書き換え
- そもそも読み取りを難しくする
  - 暗号化アルゴリズムを使用した暗号化
- 改ざんチェックを行う
  - ハッシュ値を用いた改ざんチェック
  - チェックサム

など

実行中のメモリ書き換え
- そもそも読み取りを難しくする
  - 暗号化アルゴリズムを使用した暗号化
  - メモリ上にそのまま見える値で保管せず別の値に書き換える(使用直前に戻す)
- 改ざんチェックを行う
  - ハッシュ値を用いた改ざんチェック

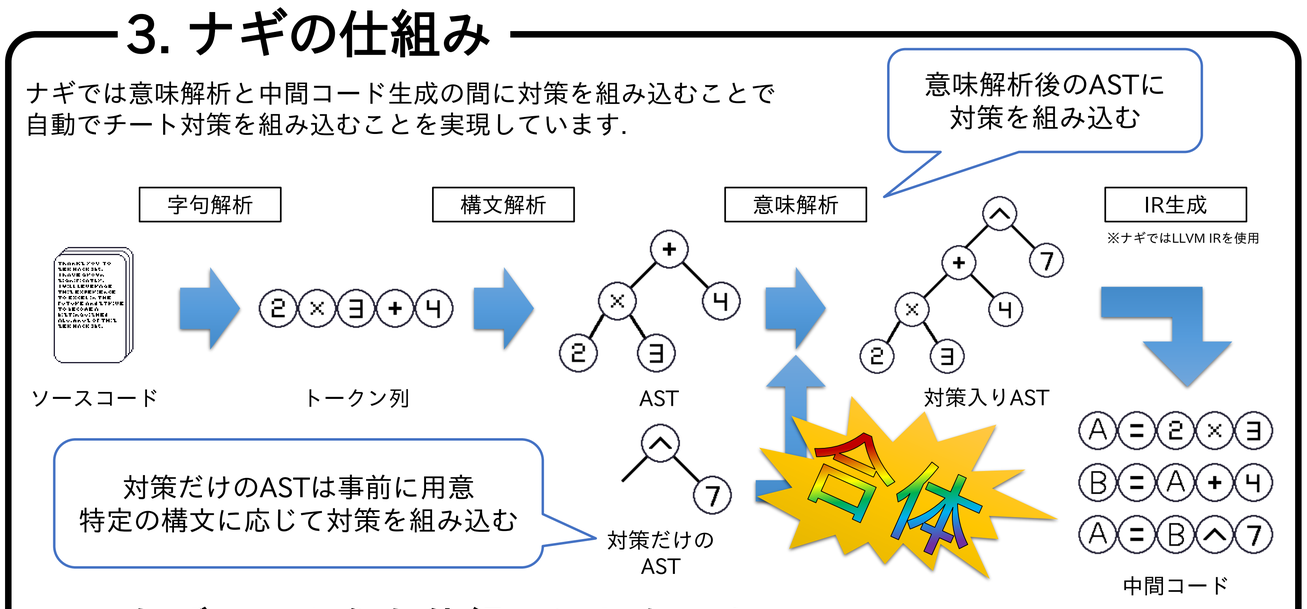
など

バイナリファイル書き換え
- 改ざんチェックを行う
  - ハッシュ値を用いた改ざんチェック
- 難読化
  - パッキング
  - ダミーの追加
  - フローの複雑化

など

3. ナギの仕組み

ナギでは意味解析と中間コード生成の間に対策を組み込むことで自動でチート対策を組み込むことを実現しています.



なぜこのような仕組みにしたのか

ナギの目的は「ゲーム開発者にチート対策を**意識させない**」ことです.
ソースコード上で何か対策を仕込む場合, 何かしらの形でソースコード上に残ってしまい「意識させない」ということに反してしまいます.
意識させないためには, 開発者の目にするソースコード上以外で組み込む必要があります.
ナギではASTの段階で対策を組み込むことでソースコードはそのまま, 内部では対策が組み込まれているといった仕組みにすることで, ゲーム開発者は意識することなくただソースコードを書くだけでよくなります.

開発者には作ることだけに集中してほしい

コンパイラの仕組み(ざっくり)

ソースコードから実行ファイル生成までの流れは字句解析, 構文解析, 意味解析, 中間コード(IR)生成の流れで行われます.

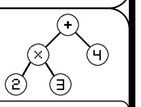
ソースコード
↓
字句解析
↓
構文解析
↓
意味解析
↓
中間コード
↓
実行ファイル

字句解析とは(ざっくり)

ソースコードを読み取り, 読みとった文字をトークンと呼ばれる最小単位に分割します
ナギではソースコード上の文字を一定の規則に従って分割しています

構文解析とは(ざっくり)

トークン列から文法に応じたASTと呼ばれる構文木を構築します
ナギではRustの文法を拝借してASTを構築しています



ASTとは(ざっくり)
AST(Abstract Syntax Tree)は抽象構文木と呼ばれ
プログラムの意味にだけ注目した構文木です
構文木にすることでプログラムが扱いやすい形になります

意味解析とは(ざっくり)

生成したASTの型情報などが正しいかチェックします

中間コード(IR)生成とは(ざっくり)

各アーキテクチャに対応したバイナリ(実行ファイル)を吐く前の状態です
処理だけに注目した形にして各アーキテクチャへの対応を行いやすい形にします
ナギではLLVM IRを使用しています

4. コード例

```
fn main() {
  let player_hp = 300;
  let enemy_hp = 999;
  let player_attack = 120;
  let enemy_attack = 50;

  while player_hp > 0 && enemy_hp > 0 {
    player_hp -= enemy_attack;
    enemy_attack -= player_attack;
  }

  if player_hp > 0 {
    print("win!");
  } else if enemy_hp > 0 {
    print("lose!");
  }
}
```

機能

四則演算

```
5 + 3
6 - 5
6 * 3
4 / 8
```

比較演算

```
6 == 1
6 != 3
6 < b
3 <= 3
3 > 6
3 >= 5
```

論理演算

```
a && b
c || d
```

変数定義

```
let a = 10;
```

関数定義

```
fn main(){
  let a = 0;
  if a == 0 {

  } else {

  }

  while a != 0 {

  }
}
```

条件分岐

```
if a == b {

} else if c != d {

} else {

}
```

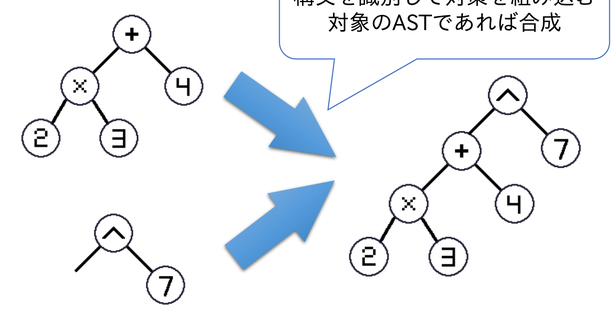
繰り返し

```
loop {
}

while a != b {

}
```

内部では

ナギのソースコード上では変数に値を代入する場合や使用する場合, 他の言語のように, 単に代入や代入した値の使用をしているように見えますが, 現在のナギでは, 変数への代入時と使用前に特定の値との XOR 演算を行っており,使用直前まで, メモリ上では異なる値になっています. 変数宣言時と変数使用時の構文の AST を識別し, それぞれに XOR 演算を付加する形で AST を合成することで実現しています.

イメージ



ソースコード上で以下のような変数Xへの代入と変数Xを使用し変数Yに2を足して代入する場合
X = 2 * 3 + 4
Y = X + 2
内部では
X = (2 * 3 + 4) ^ 7
Y = ((X ^ 7) + 2) ^ 7
のような形で表現されています

5. 他言語との比較

以下はナギの内部で行われる処理をC言語で書き直したソースコード(左側の画像)とナギのソースコード(右側の画像)です. C言語で書き直したソースコードを読むと, 実行中メモリ上を検索されることへの対策は組み込まれてはいますが, その対策によってソースコードがとても読みづらくなっていることが伝わるかと思います. ナギでは本来書きたい処理を書くだけでよいので, 対策を組み込むことで可読性が下がることを防ぎます.

Cで書き直した場合

変更された値を元に戻して計算して再度変更しているだけではあるが読みにくい…

```
int main(void) {
  int xor_value = 0x53656348616636b;
  int player_hp = xor_value ^ 300;
  int enemy_hp = xor_value ^ 999;
  int player_attack = xor_value ^ 120;
  int enemy_attack = xor_value ^ 50;

  while((xor_value ^ player_hp) > 0 && (xor_value ^ enemy_hp) > 0) {
    int player_temp = (xor_value ^ player_hp) - (xor_value ^ enemy_attack);
    int enemy_temp = (xor_value ^ enemy_hp) - (xor_value ^ player_attack);
    player_hp = xor_value ^ player_temp;
    enemy_hp = xor_value ^ enemy_temp;
  }

  if ((xor_value ^ player_hp) > 0) {
    puts("win!");
  } else if ((xor_value ^ enemy_hp) > 0) {
    puts("lose!");
  }

  return 0;
}
```

※画像サイズ調整のためinclude部分は省いています

ナギ

書きたい処理だけを書くだけ

```
fn main() {
  let player_hp = 300;
  let enemy_hp = 999;
  let player_attack = 120;
  let enemy_attack = 50;

  while player_hp > 0 && enemy_hp > 0 {
    player_hp -= enemy_attack;
    enemy_attack -= player_attack;
  }

  if player_hp > 0 {
    print("win!");
  } else if enemy_hp > 0 {
    print("lose!");
  }
}
```

疑問など

- 言語でなくともライブラリでよかったのでは
  - ライブラリだと対策を組み込もうとしていると意識させてしまい「意識させない」というコンセプトに反してしまうため
  - 独自の構文や演算子などを定義する予定
- ナギ以外の言語で書けるようにはならないのか, ゲームエンジンに組み込みたい
  - 他言語へトランスパイルを考えています
- 自動で対策が組み込まれることによって開発者の意図しない挙動にならないのか
  - 対策を組み込むことによる実行速度低下などにはどうしてもなってしまうので, 開発者が対策を組み込む箇所を指定, 対策を書き換えできるようにしたいとは考えています
- どうして自作言語?
  - 作りたかったから

今後の展望

- ゲームを楽に作るための機能の追加
  - グラフィカルな要素の実装
  - 三角関数やベクトル, 行列などの計算機能の実装
- 対策のバリエーションを増やす
  - チート行為の例で挙げた対策例
  - ネットワーク通信
  - 対策の範囲を開発者が選択できるように
- チート行為を検知した場合の対応
  - コールバック関数を呼び出すなど
- 所有権システム
- 他言語へのトランスパイル

NICT National Institute of Information and Communications Technology

本作品の作成にあたりお世話になりました, トレーナーの川合秀実氏, アシスタントの八木橋拓之氏, SecHack365に関わる皆様に感謝申し上げます