Type-safe C Compiler 構想の関連研究について

March 30, 2001 (for AMO meeting)
Yutaka Oiwa

今日の内容

- Type-Safe C Compiler (仮称) の構想
- 関連研究の brief survey
  - Safe-C
  - CodeCenter (Saber-C) [Kaufer et.al. 1988]
  - Purify [Hastings and Joyce 92]
  - StackGuard [IBM]
- 進捗: 実装の基本方針

背景: C 言語と Security Holes (1)

- C 言語: 元 Unix Kernel 記述言語
  - 構造化言語の記述性
  - アセンブラ並みのメモリ操作柔軟性
  - アセンブラモジュールとの親和性

⇒ 日常アプリケーションの記述には
必ずしも必要のない機能ではないか？

背景: C 言語と Security Holes (2)

- Internet 時代と C 言語
  - Security Hole の温床
    - Application-level Memory Safety の欠如
    - 複雑化したアプリに記述性が追いついていない
      - オブジェクト、自動メモリ管理
      - ⇒ 複雑なポインタ操作 ⇒ 安全性の問題
  - Backward Compatibility のための存在?
    - 既存のソフトウェア
    - 既存のプログラマー

背景: C 言語と Security Holes (3)

- 解決1: 他の安全な言語を使う
  - 美しいアプローチではある
  - Not widely accepted yet
    - 貧弱な実装の問題: Java (except for servlet/RMI)
    - 言語の見た目の問題(?): Caml, Scheme, etc.
  - cf. "*NBCi shopping* uses Objective Caml extensively, in the chain of tools developed by Liquid Market to gather and build the contents of the shopping engine, as well as in the transaction engine." (from Caml Homepage)

背景: C 言語と Security Holes (4)

- 解決2: C言語を安全にしよう
  - 若干後ろ向きっぽいが現実的
  - 可能か?
    - 自由なポインタ演算のエンコード
    - 整数とポインタとの相互変換
    - 既存のコードに対する対処
    - (効率至上主義者との折り合い)

背景: C 言語と Security Holes (5)

- 既存研究は?
  - 1988年頃盛んに研究された
  - デバッグツールとしてはそれなりに有用
  - 安全性保証の面では不完全
    - Untyped protection (メモリ範囲チェック)
      - メモリ上のポインタは保護できない
      - メモリ上のポインタを壊せばなんでもできる
        - C++ のメソッドテーブルは?

Type-Safe C Compiler Project

- 完全に型安全な C の処理系を作る
  - 型を意識したメモリ保護
    - 絶対にメモリ上のポインタをアクセスさせない
    - 整数から自由なポインタを作らせない
  - きちんとしたメモリ管理
    - Free したオブジェクトへのアクセスを排除
  - 既存のプログラムへの影響を最小限にする
  - そこそこの実行効率

"Benefits"

- メモリ関連の Security Hole の根絶
  - CERT Advisory の 50%は Buffer Overrun 関係の問題らしい
  - 複雑な attack (malloc-free etc.) にも対応
- OS の Process Isolation が不要に
  - 安全な動的ロードモジュールへの応用
  - メモリ保護操作の省略による高速化
  - Typed-CPU Project にも貢献できる?

既存研究の紹介

用語の定義

- Pointer access error
  - Spatial Error: メモリ空間方向のエラー
    - 配列その他の範囲外アクセス
      1. 確保されていない領域へのアクセス (Easy)
      2. 他のオブジェクトの領域へのアクセス (Difficult)
  - Temporal Error: 時間方向のエラー
    - 解放済み領域へのアクセス (Easy)
    - 再利用されたメモリ領域への古いポインタでのアクセス (Difficult)

用語の定義 (2)

- Leak Detection
  - 誤って free されていない領域の検出
- Pointer Safety
  - 任意の既存オブジェクトへのポインタを作成できるか
  - 任意の未確保領域への有効なポインタを作成できるか

Purify

- 市販のソフトウェアテストツール
  - Purify: Fast Detection of Memory Leaks and Access Errors.
    - Reed Hastings and Bob Joyce (Pure Software Inc.)
    - USENIX Winter '92, pp125-136
  - (現在は Rational Software Corp.)

Purify の基本方針

- *.o ファイルの後処理で実現
  - 全メモリアクセス命令に check code 挿入
  - ワード単位でアクセス制御
  - 独自の malloc, free によるメモリリーク検出

Purify のメモリ管理 (1)

- 各メモリワードに 2 bits の状態
  - Unallocated (-), Uninit'ed (W), Init'ed (RW)
- 各オブジェクトの周りに緩衝領域
  - 常に Unallocated 状態: バッファ溢れを検出
  - 標準値: 前 16 bytes, 後 28 bytes
  - スタック上のオブジェクトや static data も同様に隔離
    - x = malloc(100); x[5000] = 1; は検出できない

Purify のメモリ管理 (2)

- Free'ed Memory の扱い
  - Memory ageing: 解放されたメモリを一定期間再利用せずに保持しておく
    - false pointer による誤アクセスを減らす
    - 一定期間 = その後呼ばれた free の回数
    - 欠点: プログラムが激しく alloc-free を繰り返すほど保護効果が弱くなる…
      - User-defined policy-based security suffices?

Purify: performance

Normal/Purified	gcc	X11R4 maze
Run Time (s)	26/81	117/178
a.out size (kB)	815/1570	674/931
Max heap (kB)	1486/1775	540/608
Link time (s)	7/35	5/24

- gcc の実行時間で約 3.1 倍

Purify: まとめ

- Spatial Error:
  - 未確保領域: ○
  - 他のオブジェクト: ×
- Temporal Error:
  - 既解放領域: ○
  - 再利用: ×
- Leak Detection: ○
- Pointer Safety: △ (access check enforced)

Saber-C

- インタプリタベースの debug 環境
  - Saber-C An Intepreter-based Programming Environment for the C Language.
    - Stephen Kaufer, Russell Lopez, Sasha Pratap (Saber Software Inc.)
    - USENIX '88 Summer, pp161-171
  - 現名称: CodeCenter

Saber-C: 概要

- C のソースを読み込み中間コードをインタプリタ実行
  - オブジェクトファイルと組み合わせて部分デバッグ可能
  - 実行速度: 1/200 程度
- 未初期化メモリ読み込みなどを検出

Saber-C: メモリ管理

- 独自のメモリ管理システム
  - メモリレイアウトは通常実行と同じ
  - 各メモリワードにサイズと型情報を付与
    - ポインタの型情報はどうも <pointer> に見える…
  - インタプリタからのアクセス時に範囲外アクセス、未初期化メモリへのアクセスなどをチェック
    - 未初期化検知は magic value を書いておくらしい (Purify の論文より)

Saber-C: 欠点

- むちゃくちゃ遅い
- オブジェクトファイルから読み込まれたコードは no-check で実行
  - sprintf の範囲溢れなども検出できない
- Temporal Error (reuse) のチェックはできなさそう

Saber-C: まとめ

- Spatial Error:
  - 未確保領域: ○ (all code must interpreted)
  - 他のオブジェクト: △ (×?)
- Temporal Error:
  - 既解放領域: ○ (all code must interpreted)
  - 再利用: ×
- Leak Detection: ×
- Pointer Safety: △ (access check enforced)

Safe-C

- Safe Pointer を使った安全性の保証
- 今回紹介する中では唯一 heap の temporal error をまともに検出
  - Efficient detection of all pointer and array access errors.
    - Todd M. Austin, Scott E. Breach, Gurindar S. Sohi (Univ. of Wisconsin-Madison).
    - Proc. of SIGPLAN PLDI '94.

Safe-C のポインタ

- 次の要素をもつ Tagged-Pointer
  - value: 値をさす実際のポインタ
  - base: 配列先頭へのポインタ
  - size: base の指している領域のサイズ (bytes)
  - storageClass: 領域の種類
    - Heap, Local, Global
  - capability: 領域の "ID" (後述)

Safe-C: "Capability" (1)

- 各メモリ領域に一意な Capability を付与
  - Malloc, Stack frame 確保で割り当て
- 現在生きている領域の集合を管理
  - Malloc, Stack frame 確保で追加
  - Free, 関数の終了時に削除
- メモリアクセスごとに Capability check
  - ポインタの Capability が上記の集合にあるか

Safe-C: "Capability" (2)

- 特殊な Capability
  - FOREVER: 常に有効
    - Global Object に対して付与
  - NEVER: 常に無効
    - NULL pointer (0) に対して付与
- Setjmp/longjump 対策
  - Stack Frame の capability は depth-first order
    - Non-local exit: 現在と戻り先の間の cap. を削除

Safe-C: まとめ

- Spatial Error:
  - 未確保領域: ○
  - 他のオブジェクト: ○
- Temporal Error:
  - 既解放領域: ○
  - 再利用領域: ○
- Leak Detection: ×
- Pointer Safety: × (check can bypassed)

Bound Checking C Compiler

- GCC 改造による配列範囲チェック
  - A bounds checking C Compiler.
    - Richard W.M. Jones, Paul Kelly, Naranker Dulay (Imperial College (UK)).
    - Tech. Report? (1995).

BCCC: 概要

- GCC フロントエンド改造による コード挿入
  - 構文木からポインタ操作を抽出して 関数呼び出しに置き換え
  - メモリアクセス時にポインタが 確保されたメモリ領域を指しているか検査
- メモリ管理システムを変更
  - 確保されたメモリ領域の集合を tree で保持

BCCC: コード挿入

- ポインタに対する各演算に対応した関数を定義
  - + -(int) * [] -> -(ptr) < > <= >= == != ++(pre/post) --(pre/post) !
  - *, [], -> で指示先領域の生死を検査
  - Cast については記述なし

BCCC: メモリ管理

- malloc: 確保されたメモリ領域を集合に追加
  - 前後に若干の緩衝領域
- free: 集合から領域を削除
  - 領域の再利用: option1: "しない" option2: "する"…
  - 再利用された領域に対する care なし

BCCC: Performance

- パラメータ変更に対する詳細なデータ
  - Tree の深さ、探索コスト、その他…
  - でも結局何%遅いのか不明 (^^;

BCCC: まとめ

- Spatial Error:
  - 未確保領域: ○
  - 他のオブジェクト: ×
- Temporal Error:
  - 既解放領域: ○
  - 再利用: × (no-free semantics not acceptable)
- Leak Detection: ×
- Pointer Safety: △ (access check enforced)

Various replacements for malloc()

- malloc の置換ライブラリ
  - Leak Detection: dmalloc etc.
  - Garbage Collection: Bohem etc.
  - 安全性の保証は不可能

Malloc() replacements: まとめ

- Spatial Error:
  - 未確保領域: ×
  - 他のオブジェクト: ×
- Temporal Error:
  - 既解放領域: × (△: GC-based)
  - 再利用: × (△: GC-based)
  - Leak Detection: ○
- Pointer Safety:×

StackGuard etc.

- Stack buffer overrun による security hole に限定した対策
  - スタック上の変数と return address の間に書き潰し検出用の magic number を挿入
  - Low overhead expected
  - その他のエラー、飛び越し書き込みには無力

StackGuard etc.: まとめ

- Spatial Error:
  - 未確保領域: △ (stack sequencial access only)
  - 他のオブジェクト: ×
- Temporal Error:
  - 既解放領域: ×
  - 再利用: ×
  - Leak Detection: ×
- Pointer Safety: ×

既存研究のまとめ (1)

検出できるエラーの種類	空間		時間		洩検出	型安全
	未	他	解	再		
Purify	○	×	○	×	○	△
Saber-C	○	△	○	×	○	△
Safe-C	○	○	○	○	×	×
Bound-Checking C Comp.	○	×	○	×	○	△
malloc repacements	×	×	△	△	×	×
StackGuard	△	×	×	×	×	×

既存研究のまとめ (2)

- Scheme, ML レベルの安全性はない
  - メモリに格納された Pointer、あるいはポインタへのポインタに対する言及がほとんどないのはなぜ?
    - C++ のメソッドテーブルは関数ポインタの配列
      - C++ オブジェクトには関数ポインタのポインタ
    - 関数ポインタの破壊 = 任意のコード実行
      - 特にメソッドテーブルは危険
  - 主に debug に注目: 悪意の攻撃者は未想定

現在の進捗状況

ご意見募集中!!

基本方針

- 完全なメモリアクセス違反の検出
  - 配列範囲外のメモリアクセス
  - 不正なポインタ操作
    - void *, "A * → int → A *" 変換 を含む
- Well-known hacks のサポート
  - 可変サイズ構造体
    - structure S { ...; char buf[1]; }
  - 共通の構造をもつ構造体間の cast
    - ex. SOCKADDR, SOCKADDR_IN, SOCKADDR_UN

データ表現 (1)

- 各メモリ領域にサイズと型情報を付与
  - Array-boundary check
  - 実行時型情報に基づく動的なアクセス検証
  - Non-escaping local object は解析と最適化でなんとかなるはず

データ表現 (2)

- Pointer: 2-word 表現
  - <Base, offset> pair
    - サイズ情報は Base から簡単に取得可能
      - Common-subexpression elimination で最適化可能
  - Casted-pointer のためのフラグ?
    - cast されていない→サイズチェックだけで OK
    - cast されている→型情報を使った動的検証
      - ポインタの type safety を守れ!

データ表現 (3)

- Integer: やっぱり 2-words 表現
  - C 言語の保証事項:
    - void * の情報を収容できる整数型がある
    - 無変更なら上の結果の整数を void * にできる
      - pointer safety と相反 → 隠し情報を付与
  - Taura-san's suggestion
  - 最適化:
    - 「ML とかの boxed/unboxed repl. に対する最適化方針を使えるのでは?」by Sumii-san

データ表現 (4)

- メモリ管理: Garbage Collection?
  - 既存研究の方針との比較:
    - Free しない方針は論外
    - aging も論外
    - Capability-management
      - アクセスのたびに Hash は重い
      - 1-word 余計に使う
      - 2^{32} 以上のオブジェクト生成
  - Memory Leak に対する Protection にもなる

実装

- 実装言語: おそらく OCaml :-)
  - Sumii さんと "Pair-Research" 中
  - とりあえず OCaml でパーサ作成
  - 7月くらいには fib が動くくらいにはしたい?