Stack smashing analysis by abstract interpretation of binary code

Clément Ballabriga [1], **Julien Forget** [1], Guillaume Person [1]

[1]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, Lille, France
firstname.lastname@univ-lille.fr

CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille

Université de Lille

Inria

Outline

Stack smashing: vulnerable program

Example

```
#define MAX 12
void foo(char *bar)
{
   char c[MAX];
   strcpy(c, bar); // unsafe
}
int main(int argc, char **argv)
{
  foo(argv[1]);
  return 0;
}
```

Stack smashing: exploit

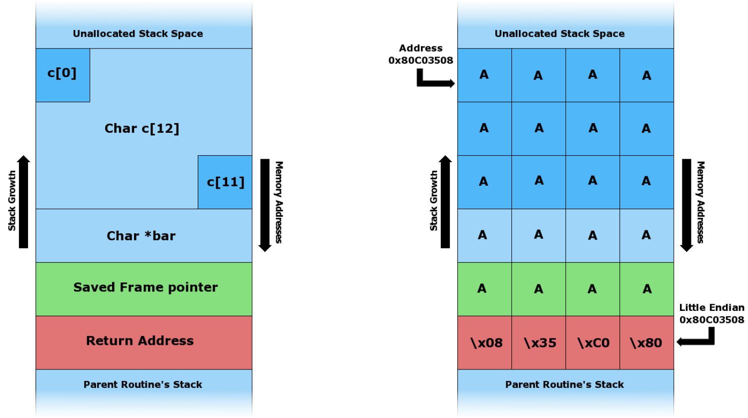

Figure: foo("AAAAAAAAAAAAAAAAAAAA\x08\x35\xC0\x80",24)

(Source Wikipedia)

Our approach: static analysis of binary code

We analyze the **binary code**:

- **Pros**
  - Can analyze closed-source programs;
  - No assumptions required about the compiler;
- **Cons**
  - Missing information:
    - No types;
    - No variables;

    $\Rightarrow$ Program accesses **data locations**: registers, memory addresses;

    $\Rightarrow$ Not your classic Abstract Interpretation;
  - Must handle different CPU instruction sets;

    $\Rightarrow$ More tedious tooling.

Our contribution

1. Abstract Interpretation:
   - Of binary code;
   - With a **relational** abstract domain;
     - $\Rightarrow$ Supports statically unknown addresses.
2. AI-based analysis to prove the **absence** of return address corruption:
   - Track function return addresses in the program abstract state;
   - Fully-automated analysis.

Outline

Reminder on abstract interpretation

Example

```
int f(int s) {
  int x=4,y=3,s,o;

  if(s)
    o=x+y;
  else o=x-y;
  // State here?
  return o;
}
```

- Two possible **concrete states** at end of function: $\{x = 4, y = 3, s = 0, o = 1\}$, $\{x = 4, y = 3, s \neq 0, o = 7\}$
- A valid **abstract state**: $\{x = 4, y = 3, 1 \leq o \leq 7\}$
- Properties proved on abstract state hold for any concrete state;
  - e.g. here we can prove that $o > 0$ at end of function.

POLYMAP, an abstract domain for binary code

With **POLYMAP**, we represent an abstract state as $(\langle c_1, \ldots, c_n \rangle, \mathcal{R}^\sharp, *^\sharp)$:

- State variables (a.k.a dimensions) are added/removed as the analysis progresses;
- $\langle c_1, \ldots, c_n \rangle$: constrains values of data locations (polyhedron);
- $\mathcal{R}^\sharp$, **register mapping**: maps polyhedra variables to registers;
- $*^\sharp$, **memory mapping**: tracks addresses $\mapsto$ values relationships.

Tracking register contents

Example

```
(0)
SET r1, #1          (1)
ADD r1, r1, #1      (2)
```

- (0): $(\top, \emptyset, \emptyset)$
- (1): $(\langle x_1 = 1\rangle, \{r_1 : x_1\}, \emptyset)$
- (2): $(\langle x_1 = 1, x_2 = x_1 + 1\rangle, \{r_1 : x_2\}, \emptyset)$
  - $\Rightarrow$ We can remove x_1: $(\langle x_2 = 2\rangle, \{r_1 : x_2\}, \emptyset)$.

Tracking memory contents

Example

```
SET r3, #42             (1)
STORE r3, [sp + #4]     (2)
```

- (1) $(\langle x_1 = 42 \rangle, \{r_3 : x_1, sp : x_2\}, \emptyset)$
- (2) $(\langle x_1 = 42, x_3 = x_2 + 4, x_4 = x_1 \rangle, \{r_3 : x_1, sp : x_2\}, \{x_3 : x_4\})$
  - $\Rightarrow$ $*(x_3) = x_4$
  - $\Rightarrow$ Address $sp + 4$ contains value 42.

Abastract interpretation procedure: main difficulties

- **Aliasing**: two different variables corresponding to the same address
  - Impacts the interpretation of LOAD and STORE;
- **Unification**: 2 different variables in 2 different states corresponding to the same location:
  - When joining the states of two program branches, unify their mappings before joining the constraints.

For details

C. Ballabriga, J. Forget, L. Gonnord, G. Lipari, and J. Ruiz. "Static analysis of binary code with memory indirections using polyhedra." In *VMCAI'19*.

Outline

Overview

Track more information during AI:

- Identify variables corresponding to return addresses;
- Track such variables for functions of the current call stack;
- Compare constraints at function call vs at function return.

Tracking return addresses

Our tool targets ARM:

- Return addresses are stored in the **link register** (LR);
- We consider:
  - Variable lr_{call} mapped to LR at function call;
  - Variable lr_{ret} mapped to LR at function return;
  - p the polyhedron at function return;

  $\Rightarrow$ Check that $p \sqsubseteq_{\diamond} \langle lr_{call} = lr_{ret} \rangle$.
- Abstract state stores a stack of live return address variables;

  $\Rightarrow$ Somehow, an **abstract shadow stack**.

Safe program

Example

```
#define MAX 12
void foo(char *bar, int n)
{
   char c[MAX];
   if (n<MAX)
     strncpy(c, bar,n); // safe
}
int main(int argc, char **argv)
{
  foo(argv[1], atoi(argv[2]));
  return 0;
}
```

- Our tool Polymalys[1] proves the **absence** of stack smashing;
- The same program with strcpy instead cannot be proved safe.

[1] https://gitlab.cristal.univ-lille.fr/otawa-plugins/polymalys

Outline

Summary

- Abstract interpretation of binary code;
  - Relevant memory addresses discovered during analysis;
  - Supports statically unknown memory addresses;
- Stack smashing detection;
  - Proves the absence of vulnerabilities
  - Fully automated;
- Limitations:
  - False negatives: invulnerable programs deemed vulnerable;
  - Scalability: AI with polyhedra=high complexity.