Binary Verification
Via Code Abstraction

FREKK VERBEEK, ABHIJT BHARADWAJ, JOSHUA BOCKENEK,
IAN ROESSLE, BINOY RAVINDRAN

NOVEMBER, 2018
Black-box Binary Verification

**Context**
- Source code is not available (legacy code, proprietary components)
- Minimizes TCB
- No semantics of source language required

**Challenges**
- Gap binary <-- formal specification
  1.) one single flat memory space
  2.) no datatypes
  3.) small step instruction semantics
- Applicability:
  1.) x86-64
  2.) real unmodified production code
- Scalability
**Formal World:**

lemma append_take_drop_id:
  fixes x :: 'a list" and n :: nat
  shows "take n x @ drop n x = x"
  by (induct n arbitrary: x, auto)

---

**Binary:**

```
491 .label_34:
492  mov  rax, qword ptr [rbp - 8] # Size:4, Opcode: 0xb,0x00,0x00
493  mov  rdx, qword ptr [rax] # Size:8, Opcode: 0xb,0x00,0x00
494  mov  rax, qword ptr [rbp - 8] # Size:4, Opcode: 0xb,0x00,0x00
495  mov  eax, dword ptr [rax + 0x10] # Size:3, Opcode: 0xb,0x00,0x00
496  mov  eax, eax # Size:2, Opcode: 0xb,0x00,0x00
497  add  rdx, rax # Size:3, Opcode: 0xb,0x00,0x00
498  mov  rax, qword ptr [rbp - 0x18] # Size:4, Opcode: 0xb,0x00,0x00
499  mov  qword ptr [rax], rdx # Size:8, Opcode: 0xb,0x00,0x00
500  mov  rax, qword ptr [rbp - 0x18] # Size:4, Opcode: 0xb,0x00,0x00
501  mov  edx, dword ptr [rbp - 0xc] # Size:3, Opcode: 0xb,0x00,0x00
502  mov  dword ptr [rax + 8], edx # Size:3, Opcode: 0xb,0x00,0x00
503  mov  rax, qword ptr [rbp - 8] # Size:4, Opcode: 0xb,0x00,0x00
504  mov  edx, dword ptr [rax + 0x10] # Size:3, Opcode: 0xb,0x00,0x00
505  mov  eax, dword ptr [rbp - 0xc] # Size:3, Opcode: 0xb,0x00,0x00
506  add  edx, eax # Size:2, Opcode: 0xb,0x00,0x00
507  mov  eax, eax # Size:5, Opcode: 0xb,0x00,0x00
508  mov  eax, 0 # Size:5, Opcode: 0xb,0x00,0x00
```
**Code Abstraction**

**Formal World:**

```haskell
record sslReader =
  buf :: "32 word list option"
  offset :: "nat"

record state =
  in_v :: sslReader
  out_v :: sslReader

\( \lambda \sigma \sigma' . \text{buf (out}_v \sigma') = \text{Some (drop (offset (in}_v \sigma))} \\
  (\text{the (buf (in}_v \sigma))})))
```

**Binary:**

```
491 .label_34:
492  mov  rax, qword ptr [rbp - 8] # Size:4, Opcode: 0xb8,0x00,0x00,0x00,
493  mov  rdx, qword ptr [rax] # Size:8, Opcode: 0xb8,0x00,0x00,0x00,
494  mov  rax, qword ptr [rbp - 8] # Size:4, Opcode: 0xb8,0x00,0x00,0x00,
495  mov  eax, dword ptr [rax + 0x10] # Size:3, Opcode: 0xb8,0x00,0x00,
496  add  rdx, eax # Size:4, Opcode: 0x89,0x00,0x00,0x00,
497  mov  rax, qword ptr [rbp - 0x18] # Size:8, Opcode: 0xb8,0x00,0x00,0x00,
498  mov  qword ptr [rax], rdx # Size:8, Opcode: 0x89,0x00,0x00,0x00,
499  mov  rax, qword ptr [rbp - 0x10] # Size:4, Opcode: 0xb8,0x00,0x00,0x00,
500  mov  edx, dword ptr [rax - 0xc] # Size:3, Opcode: 0xb8,0x00,0x00,
501  mov  dword ptr [rax + 8], edx # Size:4, Opcode: 0x89,0x00,0x00,0x00,
502  mov  rax, qword ptr [rbp - 8] # Size:4, Opcode: 0xb8,0x00,0x00,0x00,
503  mov  rdx, qword ptr [rax + 0x10] # Size:8, Opcode: 0xb8,0x00,0x00,0x00,
504  mov  edx, dword ptr [rax - 0xc] # Size:3, Opcode: 0xb8,0x00,0x00,
505  mov  eax, dword ptr [rbp - 0x10] # Size:4, Opcode: 0xb8,0x00,0x00,0x00,
506  add  edx, eax # Size:4, Opcode: 0x89,0x00,0x00,0x00,
507  mov  rax, qword ptr [rbp - 8] # Size:8, Opcode: 0xb8,0x00,0x00,0x00,
508  mov  edx, dword ptr [rax + 0x10], edx # Size:3, Opcode: 0xb8,0x00,0x00,0x00,
509  mov  eax, 0 # Size:4, Opcode: 0xb8,0x00,0x00,0x00,
```
Gap Formal World $\leftrightarrow$ Binary

**Formal World:**

\[
\neg B(\sigma) \\
\text{while } B \text{ do } f \od (\sigma, \sigma) \\
B(\sigma) \quad f(\sigma, \sigma') \quad \text{while } B \text{ do } f \od (\sigma', \sigma'') \\
\text{while } B \text{ do } f \od (\sigma, \sigma'') \\
\text{(while_base)} \\
\text{(while_rec)}
\]
Code Abstraction

- Derive abstract code from a binary with:

  - big step semantics $\leftrightarrow$ small step semantics
  - structured state / variables $\leftrightarrow$ memory / registers / flags
  - explicit control flow $\leftrightarrow$ jumps / RIP manipulation
  - compound datatypes $\leftrightarrow$
  - functions on lists / records $\leftrightarrow$ pointer arithmetic

- Establish formal refinement / abstraction relation.
- Enables application of traditional methods of software verification to binaries.
- Is already a formal proof of no unspecified behavior.
  - no null-pointer dereferences
  - no div-by-zero
  - ...
- Provides insight into semantics of binary.
  - test harnesses
  - specification inference
Code Abstraction
Simulation Relation

0. push rbp
1. mov rbp, rsp
2. mov dword ptr [rbp - 0x14], edi
3. mov qword ptr [rbp - 0x8], 1
4. mov dword ptr [rbp - 0xc], 0
5. jmp label_11

label_12:
6. shl qword ptr [rbp - 0x8], 1
7. add dword ptr [rbp - 0xc], 1
8. mov eax, dword ptr [rbp - 0xc]

label_11:
9. cmp eax, dword ptr [rbp - 0x14]
10. jb label_12
11. mov rax, qword ptr [rbp - 0x8]
12. pop rbp
13. ret

\( S \sigma_c s = (\forall i = s \vdash *[RBP s - 12, 4], \)
\( a = s \vdash *[RBP s - 8, 8]), \)
\( (\text{ret} = \text{RAX } s, ... = \sigma_c)') \)
Code Abstraction
Control Flow Graph

0. push rbp
1. mov rbp, rsp
2. mov dword ptr [rbp - 0x14], edi
3. mov qword ptr [rbp - 8], 1
4. mov dword ptr [rbp - 0xc], 0
5. jmp label_11

label_12:
6. shl qword ptr [rbp - 8], 1
7. add dword ptr [rbp - 0xc], 1
8. mov eax, dword ptr [rbp - 0xc]

label_11:
9. cmp eax, dword ptr [rbp - 0x14]
10. jb label_12
11. mov rax, qword ptr [rbp - 8]
12. pop rbp
13. ret
Code Abstraction
Basic Blocks

\[ S^C_s = (\lambda \sigma . i (\mathcal{L} \sigma') = i (\mathcal{L} \sigma) + 1 \land \\
\quad a (\mathcal{L} \sigma') = a (\mathcal{L} \sigma) \times 2) \]

\[
\begin{align*}
\text{label}_12: & \quad \text{shl} \quad \text{qword ptr [rbp - 8]}, 1 \\
\text{7. add} & \quad \text{dword ptr [rbp - 0xc]}, 1 \\
\text{8. mov} & \quad \text{eax, dword ptr [rbp - 0xc]} \\
\text{label}_11: & \quad \text{cmp} \quad \text{eax, dword ptr [rbp - 0x14]} \\
\text{10. jb label}_12
\end{align*}
\]
Code Abstraction
Control Flow Inference

0. push rbp
1. mov rbp, rsp
2. mov dword ptr [rbp - 0x14], edi
3. mov qword ptr [rbp - 8], 1
4. mov dword ptr [rbp - 0xc], 0
5. jmp label_11

label_12:
6. shl qword ptr [rbp - 8], 1
7. add dword ptr [rbp - 0xc], 1
8. mov eax, dword ptr [rbp - 0xc]

label_11:
9. cmp eax, dword ptr [rbp - 0x14]
10. jb label_12
11. mov rax, qword ptr [rbp - 8]
12. pop rbp
13. ret

∀s · P(s) → ¬H(s) ∧ (?B_r(s) = B(R(s)))
{P ∧ ?B_r} step; run_until(H ∨ ?H’) ≤ t_a {P}
{P ∧ ¬?B_r} run_until(H) ≤ skip {Q}
{P} run_until(H) ≤ while B do t_a od {Q}
Binary verification

```
0. push    rbp
1. mov     rbp, rsp
2. mov     dword ptr [rbp - 0x14], edi
3. mov     qword ptr [rbp - 8], 1
4. mov     dword ptr [rbp - 0xc], 0
5. jmp     label_11

label_12:
6. shl     qword ptr [rbp - 8], 1
7. add     dword ptr [rbp - 0xc], 1
8. mov     eax, dword ptr [rbp - 0xc]

label_11:
9. cmp     eax, dword ptr [rbp - 0x14]
10. jb      label_12
11. mov     rax, qword ptr [rbp - 8]
12. pop     rbp
13. ret
```

```
"abstract_pow2 n ≡
(λ σ σ'. i (L σ') = 0 ∧
  a (L σ') = 1);
WHILE (λ σ . i (L σ) < n) DO
   (λ σ σ'. i (L σ') = i (L σ) + 1 ∧
    a (L σ') = a (L σ) * 2)
OD;
(λ σ σ'. ret (L σ') = a (L σ))"
```

```
{ λ σ . True}
call (abstract_pow2 n)
{ λ σ . ret (L σ) = 2 ^ (unat n) }
```
Compositionality

Requirements:
1. After a function call, **callee save registers** should not be changed.
2. A function call should not mess with the **stack frame** of the caller.
Memory (register) preservation

Under which preconditions $P$ does a byte at address $a$ remain the same?

$$\{ P \land [a, 1] = v_0 \} \Rightarrow [a, 1] = v_0$$
Supported features

- Nested loops with break / continue / return
- (Non-tail) recursive functions
- Function calls
- Malloc / realloc
- Various list operations
- Nested records
- Type casting
- Signed and unsigned arithmetic
Results

x86-64 machine code:
1. Comprehensive formal model of x86-64 semantics
2. Testing setup for formal semantics
3. Symbolic execution engine

## Results: Code Abstraction

<table>
<thead>
<tr>
<th>Name</th>
<th>LoC (source)</th>
<th>LoC (assembly)</th>
<th>Effort (person days)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wc</td>
<td>55</td>
<td>137</td>
<td>4</td>
<td>Word-count. Contains loops with breaks and continue.</td>
</tr>
<tr>
<td>strncmp</td>
<td>16</td>
<td>46</td>
<td>3</td>
<td>String comparison. Contains a loop with break and return.</td>
</tr>
<tr>
<td>factorial</td>
<td>16</td>
<td>46</td>
<td>7</td>
<td>Factorial. Contains non-tail-recursion.</td>
</tr>
<tr>
<td>build_array</td>
<td>22</td>
<td>38</td>
<td>2</td>
<td>Running example. Contains malloc and an array.</td>
</tr>
<tr>
<td>stack</td>
<td>45</td>
<td>128</td>
<td>3</td>
<td>Stack data structure implemented in C++.</td>
</tr>
<tr>
<td>NSS</td>
<td>366</td>
<td>1065</td>
<td>28</td>
<td>Functions from the NSS framework Of Mozilla Firefox, containing structs passed by reference, various function calls, malloc, realloc, and various types of loops.</td>
</tr>
</tbody>
</table>
Results: Memory Preservation

Verifying the Hermitcore unikernel

<table>
<thead>
<tr>
<th>Functions</th>
<th>Count</th>
<th>SLOC (C/asm)</th>
<th>Loops</th>
<th>Recursion</th>
<th>Pointer args</th>
<th>Globals</th>
<th>Subcalls</th>
<th>-O3 done</th>
</tr>
</thead>
<tbody>
<tr>
<td>dequeue_*</td>
<td>3</td>
<td>54/155</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>buddy_*</td>
<td>4</td>
<td>50/185</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Partially</td>
</tr>
<tr>
<td>task_list_*</td>
<td>3</td>
<td>91/159</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>vring_*</td>
<td>3</td>
<td>59/97</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>string.h</td>
<td>6</td>
<td>83/358</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tasks.c</td>
<td>12</td>
<td>191/807</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>syscall.c</td>
<td>11</td>
<td>203/593</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Conclusion

Binary verification via code abstraction

- Allows use of software verification tools (SPIN, ITP with Hoare logic) to verify binaries
- Easy verification of simple properties, interactive verification of functional correctness
- Interactive but reasonably scalable
- Applicable to real production code on x86-64

Ongoing Work

- More automation
- More scalability
- More patterns corresponding to optimized code
- Termination
- Concurrency support