BAP: A Binary Analysis Platform

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz
Carnegie Mellon University

Alec Bernardi
Khalil Fleming
Overview

Program analysis platform for binary code

- No source required
- Reverse engineering
- Type inference in assembly
- Security research

Written in OCaml

- Additional C & Python bindings

http://bap.ece.cmu.edu
Background

Approach:
1. Disassemble binary
2. Lift to intermediate language
3. Perform analysis

Difficulties:
- Side-effects
- Control flow
- Formal semantics

Other platforms:
- CodeSurfer/x86
- McVeto
- Phoenix
- Jakstab

Other binary tools:
- DynInst
- Valgrind

```bash
1  add %eax, %ebx  # ebx=eax+ebx (sets OF, SF, ZF, AF, CF, PF)
2  shl %cl, %ebx  # ebx=ebx<<cl (sets OF, SF, ZF, AF, CF, PF)
3  jc target     # jump to target if carry flag is set
```
Background - asmc2

Approach:
1. Decompile x86 to C
2. Perform program analysis

Drawbacks:
- Implicit side-effects
- Difficult control flow analysis
- Written in C++
Background - Vine

Approach:
1. Disassemble x86
2. Lift to VEX intermediate language
3. Augment to expose side effects
4. Perform program analysis

Drawbacks:
- No IL formal semantics
- No bi-endian architectures (e.g. ARM)
Introducing BAP

Goals:
- Improve intermediate language
  - Formal semantics
  - Explicit assembly side-effect representation
  - Model of bi-endian memory operations
- Analysis and verification techniques
  - Include useful standards
  - Allow user-defined analyses

Approach:
1. Disassemble binary
2. Lift to side-effect free BAP IL
3. Perform program analysis

Table 1. An abbreviated syntax of the BAP IL

\[
\begin{align*}
\text{program} & ::= \text{stmt*} \\
\text{stmt} & ::= \text{var} ::= \text{exp} | \text{jmp exp} | \text{cjmp exp,exp,exp} | \text{assert exp} \\
& \quad | \text{label label\_kind} | \text{addr address} | \text{special string} \\
\text{exp} & ::= \text{load(exp, exp, exp, } \tau_{\text{reg}} \text{)} | \text{store(exp, exp, exp, exp, } \tau_{\text{reg}} \text{)} | \text{exp } \diamond_{\text{u}} \text{ exp} \\
& \quad | \text{let var = exp in exp} | \text{unknown(string, } \tau \text{)}
\end{align*}
\]
Architectural Overview

Frontend
- Linear-sweep disassembly algorithm
- Lifts instructions into BAP IL

Backend
- Optimization
- Representation transformation (optional)
- Program analysis
Architectural Overview - Included Analyses

Built-in analyses

- Slicing source/sink pairs
  - Only consider affecting statements (forward)
  - Only consider affected statements (backwards)

- IL optimization
  - 4.5x speedup
  - Solves 81% of otherwise unsolvable VCs

- IL evaluation
  - Record control flow
  - Randomized testing
  - Verify IL semantics against real program
Architectural Overview - Verification Conditions

Provided VC generation methods:
- Dijkstra’s weakest preconditions
  - VC is $O(2^n)$
- Flanagan & Saxe’s algorithm
  - VC is $O(n^2)$
- Forward-direction variant of F&S’s algorithm
- Forward symbolic execution
Applications

TIE: Principled Reverse Engineering of Types in Binary Programs
JongHyup Lee, Thanassis Avgerinos, and David Brumley
Carnegie Mellon University

Efficient Directionless Weakest Preconditions
David Brumley and Ivan Jager
Carnegie Mellon University

Binary-only symbolic execution
- Patch-based exploit generation
- Malware analysis

http://bitblaze.cs.berkeley.edu
http://security.ece.cmu.edu
Limitations

Supports subset of assembly features
- No floating point instructions
- No privileged instructions

Proof of correctness
- No formal semantics for x86
- Randomized testing

Lifting expects aligned instruction sequence
- User must identify code locations
- By hand, symbol data, or recursive descent analysis

Dynamic code
- Execution trace feature
- Unresolved indirect jumps
- Symbolic execution
Questions?