Experiences with the Carnegie Mellon Binary Analysis Platform (CMU BAP)

Sam L. Thomas, CNRS, IRISA
sam.thomas@irisa.fr
Introduction - what is BAP?

Binary analysis framework:

- For program analysis
- For (aiding) reverse engineering (plugin for IDA similar to BinCAT\(^1\))
- Written in OCaml (with bindings for C, Python and Rust)
- Support for many architectures (ARM, MIPS, PPC, x86/x86-64)

\(^1\) [https://github.com/BinaryAnalysisPlatform/bap-ida-python](https://github.com/BinaryAnalysisPlatform/bap-ida-python)
(Very brief) project history

Reengineering of Vine\(^1\) from the BitBlaze project

Each iteration, different IR: C AST → VEX → BIR/BIL

Used by CyLab spin-off startup ForAllSecure

...third binary analysis framework by same group: asm2c → Vine → BAP

BAP itself has been re-architectured during its development:

1. Library-based
2. Plugin-based + extension points

...who produced MAYHEM (automated cyber reasoning system)
Use in research*

- **Byteweight**
  - Machine learning-based function start identification

- **MAYHEM**
  - Automated vulnerability discovery and exploit generation

- **oo7 (Spectre checker)**
  - Automated (binary-based) Spectre variant detection

- **Stringer**
  - Semi-automated backdoor & undocumented functionality detection

- **HumIDIFy**
  - Semi-automated backdoor detection (machine learning + static analysis)

- **Saluki**
  - Finding Taint-style Vulnerabilities with Static Property Checking (formal models of CWEs)

- **Moflow framework**
  - Automated vulnerability discovery and triage

* See bibliography at end of presentation for references/links
My experience with BAP

As part of PhD:

- BAP version 0.9.9
- Built two tools for (semi-)automated backdoor detection (using OCaml API):
  - Stringer (static analysis)
  - HumlDIFy (ML + static analysis)
- Used tools as part of workshop for [company] on backdoor detection
A tour of BAP*
Architecture

- Core BAP library; features implemented with plugins
- By default provides:
  - LLVM based disassembler/loader backend
  - Hand-written lifters for ARM, MIPS, PPC, x86, x86-64
  - Function start/CFG recovery
- Represents a program in an IR (BIR); components represented by “Terms”
- Terms annotated with attributes (basic blocks -- BIL)
Extensible core components

- Loader (e.g., Mach-O, etc.)
- Target (e.g., RISC-V, etc.)
- Disassembler
- Attributes (given to terms)
- Symbolizer
- Rooter
- Brancher
- (CFG) Reconstructor
- Analysis (aka pass)
BAP Instruction Language (BIL)

- High-level IL
- ML-style constructs (e.g., let bindings)
- Models side-effects (e.g., modifications to EFLAGS via add, etc.)
- Simple and human-readable
- Formally defined (operational semantics\(^1\), etc.)

\(^1\) [Link](https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.3/bil.pdf)

```assembly
0000023b:  sub call_gmon_start()
00000212:
00000214:  RSP := RSP - 8
0000021b:  RAX := mem[0x600FE0, e1]:u64
0000021c:  v303 := RAX
00000222:  ZF := 0 = v303
00000228:  when ZF goto %00000223
00000227:  goto %00000224
```

Side-effects on EFLAGS & stack modelled explicitly
void printme(const char *str) {
    puts(str);
}

0x4006ed: push rbp
0x4006ee: mov rbp, rsp
0x4006f1: mov edi, 0x4008e0
0x4006f6: call 0x400510
0x4006fb: pop rbp
0x4006fc: ret
Same example in VEX (using angr)

IRSB {
  t0:Ity_I64 t1:Ity_I64 t2:Ity_I64 t3:Ity_I64 t4:Ity_I64 t5:Ity_I64 t6:Ity_I64
  t7:Ity_I64 t8:Ity_I64 t9:Ity_I64 t10:Ity_I64 t11:Ity_I64

  00 | ------ IMark(0x4006ed, 1, 0) ------
  01 | t0 = GET:I64(rbp)
  02 | t5 = GET:I64(rsp)
  03 | t4 = Sub64(t5, 0x0000000000000008)
  04 | PUT(rsp) = t4
  05 | STle(t4) = t0
  06 | ------ IMark(0x4006ee, 3, 0) ------
  07 | PUT(rbp) = t4
  08 | ------ IMark(0x4006f1, 5, 0) ------
  09 | PUT(rdi) = 0x00000000000000004008e0
  10 | PUT(rip) = 0x00000000000000004006f6
  11 | ------ IMark(0x4006f6, 5, 0) ------
  12 | t8 = Sub64(t4, 0x0000000000000008)
  13 | PUT(rsp) = t8
  14 | STle(t8) = 0x00000000000004006fb
  15 | t10 = Sub64(t8, 0x0000000000000800)
  16 | ====== AbiHint(0xt10, 128, 0x000000000000400510) ======

  NEXT: PUT(rip) = 0x000000000000400510; Ijk_Call
}

IRSB {
  t0:Ity_I64 t1:Ity_I64 t2:Ity_I64 t3:Ity_I64
t4:Ity_I64 t5:Ity_I64 t6:Ity_I64 t7:Ity_I64

  00 | ------ IMark(0x4006fb, 1, 0) ------
  01 | t1 = GET:I64(rsp)
  02 | t0 = LDle:I64(t1)
  03 | t5 = Add64(t1, 0x0000000000000008)
  04 | PUT(rsp) = t5
  05 | PUT(rbp) = t0
  06 | PUT(rip) = 0x00000000000000004006fc
  07 | ------ IMark(0x4006fc, 1, 0) ------
  08 | t3 = LDle:I64(t5)
  09 | t4 = Add64(t5, 0x00000000000000000000000000000000)
  10 | PUT(rsp) = t4
  11 | t6 = Sub64(t4, 0x0000000000000800)
  12 | ====== AbiHint(0xt6, 128, t3) ======
  13 | NEXT: PUT(rip) = t3; Ijk_Ret
}
Plugins

- Compositional in functional sense; two variants:
  - Extensions
  - Passes (special type of extension to implement analyses)

- State of framework passed between passes
- Composition of passes enables more complex analyses
Compute ratio of “jump” terms to other BIR terms

```ocaml
open Core_kernel.Std
open Bap.Std

let counter = object
  inherit [int * int] Term.visitor
  method! enter_term _ _ (jmps,total) = jmps,total+1
  method! enter_jmp _ (jmps,total) = jmps+1,total
end

let main proj =
  let jmps,total = counter#run (Project.program proj) (0,0) in
  printf "ratio = %d/%d = %g\n" jmps total (float jmps /. float total)

let () = Project.register_pass' main
```

Object to “visit” all IL terms

State is passed as “proj” or Project in BAP nomenclature
BAP from Python

```python
import bap
from bap.adt import Visitor

class Counter(Visitor):
    def __init__(self):
        self.jmps = 0
        self.total = 0

    def enter_Jmp(self, jmp):
        self.jmps += 1

    def enter_Term(self, t):
        self.total += 1

proj = bap.run('/bin/true')
count = Counter()
count.run(proj.program)
print("ratio = {0}/{1} = {2}".format(count.jmps, count.total, count.jmps/float(count.total)))
```
Plugins - Extension points

- Extend core analysis components:
  - Handle new file formats
  - Implement new CFG recovery algorithm
  - ...

- Provides a means of testing research on different aspects of binary analysis without having to focus on other aspects:
Byteweight

- Implemented as an extension to BAP as a “rooter”
- Provides ML-based function start identification for stripped binaries
- Reported improvements over state-of-the-art (IDA Pro)

```ocaml
let main path length threshold =
let finder arch = create_finder path length threshold arch in
let find finder mem =
   Memmap.to_sequence mem |
   Seq.fold ~init:Addr.Set.empty ~f:(fun roots (mem,v) ->
      Set.union roots @@ Addr.Set.of_list (finder mem)) in
let find_roots arch mem = match finder arch with
| Error _ as err ->
   warning "unable to provide rooter service";
   err;
| Ok finder -> match find finder mem with
   | roots when Set.is_empty roots ->
      info "no roots was found";
      info "advice - check your compiler's signatures";
      Ok (Rooter.create Seq.empty)
   | roots -> Ok (roots |> Set.to_sequence |> Rooter.create) in
let rooter =
   let open Project.Info in
   Stream.Variadic.(apply (args arch $ code) ~f:find_roots) in
   Rooter.Factory.register name rooter
```

Implementation of rooter and its registration as an extension to BAP’s analysis
Primus

- Micro execution\(^1\) framework (implemented as an “analysis”)
- Start execution from anywhere (without input or test driver)
- Scriptable (Primus Lisp)

Taint

- Built as a Primus “observer”
- Abstract taint tracking engine
- Policy-based taint propagation
- Configuration via OCaml or Primus Lisp

```bash
bap ./test --taint-reg=malloc_result \
  --run \ 
  --run-entry-points=all-subroutines \ 
  --primus-limit-max-length=4096 \ 
  --primus-promiscuous-mode \ 
  --primus-greedy-scheduler \ 
  --primus-propagate-taint-from-attributes \ 
  --primus-propagate-taint-to-attributes \ 
  --print-bir-attr=tainted-{ptrs,regs} \ 
  --dump=bir:result.out \ 
  --report-progress
```

0000019d: call @malloc with return %0000019e
...
...
000001a7:
.tainted-regs {R0 => [0000019d]}
000003aa: memmove_result := R0
...
Saluki

I. Gotovchits, R. V. Tonder, D. Brumley. “Saluki: Finding Taint-style Vulnerabilities with Static Property Checking” (BAR Workshop @ NDSS), 2018

Spectre Checker

G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, A. Roychoudhury. “oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis” (preprint), 2018

https://arxiv.org/abs/1807.05843

```c
void victim_function_v01(size_t x) {
  // CB (branch)
  if (x < array1_size) {
    //IM1 (access to array1)
    //IM2 (access to array2)
    temp &= array2[array1[x] * 256];
  }
}
```
First-hand experiences
HumIDIFy

- SLOC: 3,510
- BAP used to implement static analyses:
  - Feature extraction for ML
  - To implement runtime for binary analysis DSL
- Good runtime perf: \( (1.31 + 0.291 + 1.53) \approx 3.13s \)
- Detects numerous backdoors + anomalous functionality in Linux embedded binaries

```
rule main() =
    handles_tcp()                   &&
    warn_handles_udp(handles_udp()) &&
    has_file_access()
```
Stringer

- SLOC: ≈ 3,000
- BAP used to implement static analyses:
  - Automatically identify static data comparison functions (in absence of symbols + dynamic linking)
  - Locate static data comparisons that cause execution of unique flows

<table>
<thead>
<tr>
<th>Comparison Function</th>
<th>Identification</th>
<th>Reachability Analysis</th>
<th>Report Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcmp(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strncmp(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>std::string::operator==(...)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[f] 37.66: sub_60118
34.89: 664225 (via: strcmp)
2.77: root (via: strcmp)

* Latest angr (using PyPy) takes > hour to just perform CFG recovery (CFGFast)
Using BAP for research

Pros:
❖ OCaml
❖ Documentation
❖ Support (active Gitter channel)
❖ Tutorials
❖ Fast (native code)
❖ Easy to test isolated research ideas/proof-of concepts

Cons:
❖ OCaml
❖ Steep learning curve
❖ Open-source examples
❖ Lack of visible community
❖ Fragmented contributions
Problems & Solutions

- Steep learning curve (even with substantial experience in OCaml)
- OCaml

- Byteweight for function start recovery (not perfect\(^1\))
- Interworking with ARM/Thumb executables
- CFG recovery:
  - No direct support for indirect branches

---

Problems & Solutions

❖ Interface with IDA ➔ current version supports this via plugin
  ➢ Function identification
  ➢ CFG recovery
  ➢ Symbols

❖ Pass “T” flag per block from IDA to BAP to support ARM/Thumb interworking

❖ Recent plugin implementing VSA to aid in CFG recovery¹

¹ https://github.com/draperlaboratory/cbat_tools
Conclusions
Conclusions

- Highly suited for research
- Barrier for adoption largely due to language choice (also fast moving development)
- Extensible
- Fast
References


❖ S. L. Thomas, T. Chothia, F. D. Garcia. “Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality”, ESORICS, 2017


