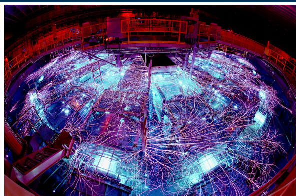
Exceptional service in the national interest



Quameleon: A Lifter and Intermediate Language for Binary Analysis

Samuel D. Pollard, Philip Johnson-Freyd, Jon Aytac, Tristan Duckworth, Michael J. Carson, Geoffrey C. Hulette, Christopher B. Harrison

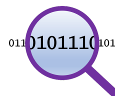
September 13, 2019

U.S. DEPARTMENT OF ENERGY NNSA National Nuclear Security Administration

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND No. SAND2019-10608 C

About Us

- About me: Ph.D. candidate at the University of Oregon, summer intern at Sandia National Labs



Digital Foundations & Maths

About Us

- About me: Ph.D. candidate at the University of Oregon, summer intern at Sandia National Labs
- The other six authors work at Sandia with some portion of their time spent on Quameleon

Introduction

- Sandia does forensic analysis of legacy, high-consequence systems

Introduction

- Sandia does forensic analysis of legacy, high-consequence systems
  - e.g. maintaining nuclear weapon control systems

Introduction

- Sandia does forensic analysis of legacy, high-consequence systems
  - e.g. maintaining nuclear weapon control systems
- Typically decades old with large portions written in assembly

Introduction

- Sandia does forensic analysis of legacy, high-consequence systems
  - e.g. maintaining nuclear weapon control systems
- Typically decades old with large portions written in assembly
- Original authors or source code may not be available

Introduction

- Sandia does forensic analysis of legacy, high-consequence systems
  - e.g. maintaining nuclear weapon control systems
- Typically decades old with large portions written in assembly
- Original authors or source code may not be available
- Our use case: analyze simple systems completely

Introduction

- Sandia does forensic analysis of legacy, high-consequence systems
  - e.g. maintaining nuclear weapon control systems
- Typically decades old with large portions written in assembly
- Original authors or source code may not be available
- Our use case: analyze simple systems completely
- Current tools do not support our architectures nor do they seem easily adapted

Introduction

- Sandia does forensic analysis of legacy, high-consequence systems
  - e.g. maintaining nuclear weapon control systems
- Typically decades old with large portions written in assembly
- Original authors or source code may not be available
- Our use case: analyze simple systems completely
- Current tools do not support our architectures nor do they seem easily adapted
- We need lifters (decompilers) and verification tools for weird ISAs

History

- Prior work consisted of one-off Haskell programs for a single ISA and single binary

History

- Prior work consisted of one-off Haskell programs for a single ISA and single binary
- Successful but not scalable

History

- Prior work consisted of one-off Haskell programs for a single ISA and single binary
- Successful but not scalable
- Rewrite started as a summer project with M6800

History

- Prior work consisted of one-off Haskell programs for a single ISA and single binary
- Successful but not scalable
- Rewrite started as a summer project with M6800
- Has since expanded to a small team working on Quameleon (the other six authors)

History

- Prior work consisted of one-off Haskell programs for a single ISA and single binary
- Successful but not scalable
- Rewrite started as a summer project with M6800
- Has since expanded to a small team working on Quameleon (the other six authors)

Motivation

- Need to analyze binaries on proprietary ISAs

Motivation

- Need to analyze binaries on proprietary ISAs
  - ISAs not supported by existing tools
  - No machine-readable specification
  - Bad old days: No IEEE-754 floats, no 8-bit bytes

Motivation

- Need to analyze binaries on proprietary ISAs
  - ISAs not supported by existing tools
  - No machine-readable specification
  - Bad old days: No IEEE-754 floats, no 8-bit bytes
- Other tools gain lots of efficiency from expressive ISAs and feature-rich ILs

Motivation

- Need to analyze binaries on proprietary ISAs
  - ISAs not supported by existing tools
  - No machine-readable specification
  - Bad old days: No IEEE-754 floats, no 8-bit bytes
- Other tools gain lots of efficiency from expressive ISAs and feature-rich ILs
- We instead require an adaptable IL

Motivation

- Need to analyze binaries on proprietary ISAs
  - ISAs not supported by existing tools
  - No machine-readable specification
  - Bad old days: No IEEE-754 floats, no 8-bit bytes
- Other tools gain lots of efficiency from expressive ISAs and feature-rich ILs
- We instead require an adaptable IL

Fun example: cLEMENCy architecture made up for DEFCON had 9-bit bytes, 27-bit words, middle-endian [3]

Design Goals of the Quameleon Intermediate Language (QIL)

- Sound analysis of binaries

Design Goals of the Quameleon Intermediate Language (QIL)

- Sound analysis of binaries
- Lift binaries into a simple IL amenable to multiple analysis backends

Design Goals of the Quameleon Intermediate Language (QIL)

- Sound analysis of binaries
- Lift binaries into a simple IL amenable to multiple analysis backends
- Closer to LLVM IR in spirit

Design Goals of the Quameleon Intermediate Language (QIL)

- Sound analysis of binaries
- Lift binaries into a simple IL amenable to multiple analysis backends
- Closer to LLVM IR in spirit
- Contrast with Ghidra [1], angr [2]: we know intended behavior

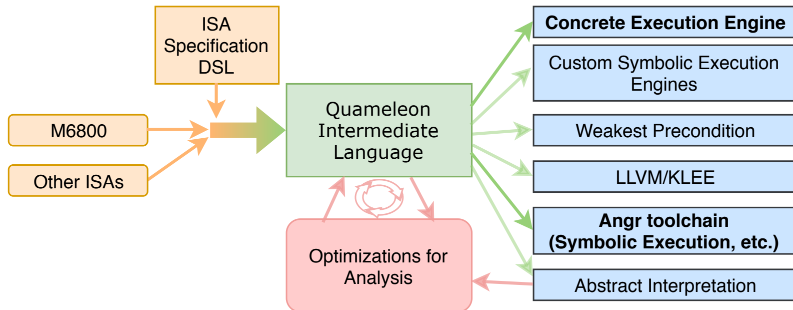
Design Goals of the Quameleon Intermediate Language (QIL)

- Sound analysis of binaries
- Lift binaries into a simple IL amenable to multiple analysis backends
- Closer to LLVM IR in spirit
- Contrast with Ghidra [1], angr [2]: we know intended behavior
- Size of QIL (~ 60 instructions) means easy to manipulate, harder to write

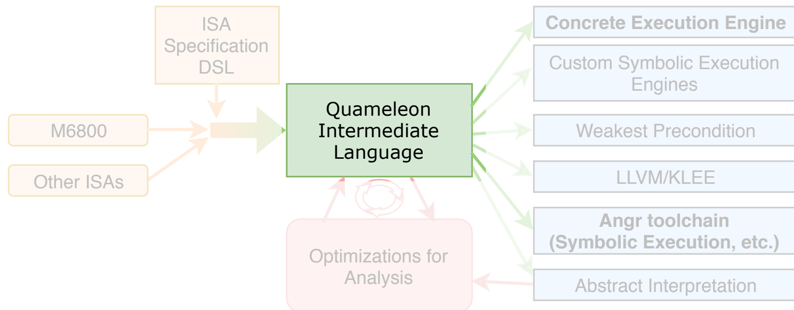
Design Goals of the Quameleon Intermediate Language (QIL)

- Sound analysis of binaries
- Lift binaries into a simple IL amenable to multiple analysis backends
- Closer to LLVM IR in spirit
- Contrast with Ghidra [1], angr [2]: we know intended behavior
- Size of QIL (~ 60 instructions) means easy to manipulate, harder to write
- Balance this with Haskell as a macro-assembler for QIL

Architectural Overview



Architectural Overview



QIL = Quameleon Intermediate Language

QIL Types

- Values: bit vectors of arbitrary width
- Locations: where values can be written
- Blocks: Single-entry, multiple exit
- Labels: Start of a block
- RAM: Mutable cells of Locations indexed by Values
- JoinPoints: Continuation within a block
- I/O: Like volatile variables

QIL Programs

A program consists of four sections:

1. Size of Locations
2. Sequence of allocations (of registers and memories)
3. Sequence of blocks, each binding a label
4. A code entry point

QIL Programs

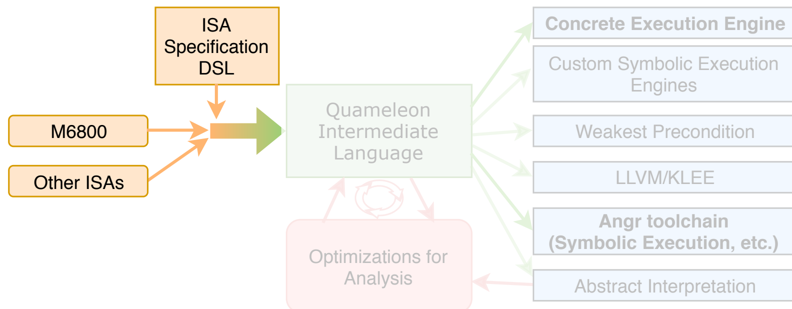
A program consists of four sections:

1. Size of Locations
2. Sequence of allocations (of registers and memories)
3. Sequence of blocks, each binding a label
4. A code entry point

Within a block

- Variables are static single assignment
- No loops

Haskell DSL



Sample M6800

```
LDA A #14 ; A <- 0xE
AND A $40 ; A <- A & [0x40]
```

We want to match the manual closely

...and Its Corresponding Semantics

```
AND r l -> do
  ra <- getRegVal r
  op <- loc8ToVal l -- Loc. of 8 bits in RAM
  rv <- andBit ra op
  z <- isZero rv
  writeReg r rv
  writeCC Zero z -- CC = Condition Code
  branch next
```

…and Its Corresponding QIL

```
1  code_ptr_size: S16
2  alloc_part: {
3     &1 := alloc[S8] // Reg A
4     &2 := alloc[S8] // Reg B
5     &3 := alloc[S16] // Reg X
6     &4 := alloc[S16] // Reg PC
7     &5 := alloc[S16] // Reg SP
8     &6 := alloc[S1] // Carry Flag
9     &7 := alloc[S1] // Overflow Flag
10    &8 := alloc[S1] // Zero Flag
11    &9 := alloc[S1] // Negative Flag
12    &10 := alloc[S1] // Interrupt Flag
13    &11 := alloc[S1] // HalfCarry Flag
14    MEM(1) := buildMemory[S16 S8]
15 }
```

...and Its Corresponding QIL (cont.)

```
16  code_part: {
17    @1 := block { }
18    @2 := registered_block "AND A (DIR8 0x40)" 2 {
19      %1 := readLoc[S8] &1 // read Register A
20      &12 := MEM(1)[S16].BV[S8](40)
21      %2 := readLoc[S8] &12
22      %3 := AndBit[S8] %1 %2
23      writeLoc[S8] &1 %3 // set Register A
24      branch @1
25    }
26    @3 := registered_block "LDA A (IMM8 14)" 0 {
27      writeLoc[S8] &1 BV[S8](e) // set Register A
28      %1 := IsZero[S8] BV[S8](e)
29      writeLoc[S1] &8 %1 // set Zero Flag
30      branch @2
31    }
32    @4 := block { branch @3 }
33  }
34  entry_point: @4
```

Backends

ISA Specification DSL

M6800

Other ISAs

Quameleon Intermediate Language

Optimizations for Analysis

Concrete Execution Engine

Custom Symbolic Execution Engines

Weakest Precondition

LLVM/KLEE

Angr toolchain (Symbolic Execution, etc.)

Abstract Interpretation

Current Backends

1. Emulator

Current Backends

1. Emulator
2. Bridge to angr
   - angr is a symbolic execution engine primarily for cybersecurity

Current Backends

1. Emulator
2. Bridge to angr
   - angr is a symbolic execution engine primarily for cybersecurity
   - Originally planned to translate from QIL to angr's IR, VEX

Current Backends

1. Emulator
2. Bridge to angr
   - angr is a symbolic execution engine primarily for cybersecurity
   - Originally planned to translate from QIL to angr's IR, VEX
   - VEX has byte-centric memory model, different functions for add32, add16, etc.

Current Backends

1. Emulator
2. Bridge to angr
   - angr is a symbolic execution engine primarily for cybersecurity
   - Originally planned to translate from QIL to angr's IR, VEX
   - VEX has byte-centric memory model, different functions for add32, add16, etc.
   - We needed addition of 96 bit integers

Current Backends

1. Emulator
2. Bridge to angr
   - angr is a symbolic execution engine primarily for cybersecurity
   - Originally planned to translate from QIL to angr's IR, VEX
   - VEX has byte-centric memory model, different functions for add32, add16, etc.
   - We needed addition of 96 bit integers
   - Easier to treat QIL as an ISA that angr can execute!

Optimizations

ISA Specification DSL

M6800

Other ISAs

Quameleon Intermediate Language

Optimizations for Analysis

Concrete Execution Engine

Custom Symbolic Execution Engines

Weakest Precondition

LLVM/KLEE

Angr toolchain (Symbolic Execution, etc.)

Abstract Interpretation

QIL-QIL Optimizations

The goal is to facilitate analysis

QIL-QIL Optimizations

The goal is to facilitate analysis

- Constant folding
- Branch to known value
- Dead code elimination

} Reduce code size

QIL-QIL Optimizations

The goal is to facilitate analysis

- Constant folding
- Branch to known value
- Dead code elimination
- Inlining with simple heuristics
  e.g. inline everywhere
- Defunctionalization

} Reduce code size (Constant folding, Branch to known value, Dead code elimination)

} Simplify CFG (Inlining with simple heuristics, Defunctionalization)

Future Work

- Jump to a Location in memory
  - Use abstract interpretation to find Locations code could jump

Future Work

- Jump to a Location in memory
    - Use abstract interpretation to find Locations code could jump
- Formalize QIL and QIL-QIL transformations in Coq

Future Work

- Jump to a Location in memory
  - Use abstract interpretation to find Locations code could jump
- Formalize QIL and QIL-QIL transformations in Coq
- Loops with statically-known bounds in blocks
  - Don't need the full sophistication of more richly-featured ILs

Future Work

- Jump to a Location in memory
  - Use abstract interpretation to find Locations code could jump
- Formalize QIL and QIL-QIL transformations in Coq
- Loops with statically-known bounds in blocks
  - Don't need the full sophistication of more richly-featured ILs
- Plan to open source as much as possible

Conclusion

- Quameleon is a tool for sound binary analysis in its early stages
- QIL is a typed, RISC-like IL to specify legacy architectures
- Leverage machine readability with the simplicity of QIL
- Leverage features of Haskell as an assembler for QIL
- Haskell DSL matches the structure of ISA specs
- Prefer the flexibility of few assumptions over efficiency of powerful model

References I

[1] National Security Agency Research Directorate.
Ghidra: A software reverse engineering (sre) framework, 2019.
Available at https://www.ghidra-sre.org.

[2] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G.
SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis.
In *IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 138–157.

[3] Trail of Bits.
An extra bit of analysis for clemency.
Available at https://blog.trailofbits.com/2017/07/30/an-extra-bit-of-analysis-for-clemency/.