SAND2019-7337C

Quameleon: A Lifter and Intermediate Language for Binary Analysis

Philip Johnson-Freyd
pajohn@sandia.gov
Digital Foundations & Mathematics, Sandia National Laboratories

Samuel D. Pollard
spolla@sandia.gov
Digital Foundations & Mathematics, Sandia National Laboratories

Jon Aytac
Digital Foundations & Mathematics, Sandia National Laboratories

Tristan Duckworth
Cyber Systems Research, Sandia National Laboratories

Michael J. Carson
Cyber Systems Research, Sandia National Laboratories

Geoffrey Compton Hulette
Digital Foundations & Mathematics, Sandia National Laboratories

Christopher B. Harrison
Cyber Systems Research, Sandia National Laboratories

Abstract

We present Quameleon, an analysis framework for low-level programs. Quameleon is takes as input a program in binary or assembly language format and translates, or "lifts" this program into an intermediate representation called the Quameleon Intermediate Language (QIL) which is then amenable for analysis. The primary features of QIL are: all operations are parametric over the bit sizes on which they operate and a value can take on any type. These allow us to keep the core language of QIL small and allow us to treat memory and register values as logical formulae, for example, in order to support symbolic execution. We discuss the design of QIL and Quameleon and how they support analysis.

CCS Concepts • **Software and its engineering → Formal software verification**.

Keywords ISA, specification, disassembly, binary analysis, verification

SpISA '19, September 13, 2019, Portland, OR

ACM Reference Format:

Philip Johnson-Freyd, Samuel D. Pollard, Jon Aytac, Tristan Duckworth, Michael J. Carson, Geoffrey Compton Hulette, and Christopher B. Harrison. 2019. Quameleon: A Lifter and Intermediate Language for Binary Analysis. In *ITP 2019: Proceedings of the Tenth Conference on Interactive Theorem Proving. September 8–13, Portland, OR.* ACM, New York, NY, USA, 4 pages.

Introduction

Quameleon is a binary analysis framework: its input is a program in binary or assembly language and its output is some high-level analysis. We accomplish this by transforming (or *lifting*) binaries into an intermediate language (IL), with which we can perform various optimizations while also providing a single interface for analysis backends. In this paper we primarily describe QIL, the Quameleon Intermediate Language (pronounced "quill").

We write the semantics of a target machine language in an embedded Haskell DSL, then generate analyzable QIL from target programs. QIL features a simple type system targeted to the domain with types for sized bit vectors, code pointers, and memory locations; polymorphism and genericity are limited to the meta (Haskell) level. QIL's most significant limitation is it assumes a Harvard architecture; code and data are separate and self modification is forbidden. We plan to address this limitation in future work.

In this work we focus more on the frontends (lifting) and the design of QIL rather than the backends (analysis). Current backends include a bridge to angr [6] and a concrete executor. Quameleon supports multiple ISAs but in this paper we pick examples entirely from the, beautifully simple, Motorola M6800 ISA.

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.

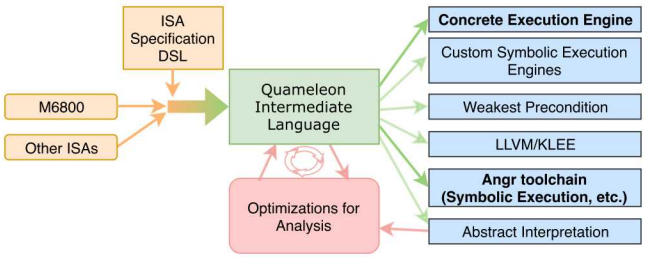

Figure 1. Quameleon pipeline. The bolded blue boxes are the currently supported backends.

Disassembly and Lifting

The first part of our binary analysis work is to *lift* a binary into an intermediate representation. Essentially, this means exposing the semantic structure of the binary at a higher level of abstraction.

Our use case is similar to tools like Ghidra [5] and IDA [2]; this means that from scratch, the process for analyzing the binary consists of three main steps.

1. Creating a machine-readable specification of the ISA. This is accomplished by creating a Haskell data type and associating a semantics with it.
2. Specifying how to disassemble individual instructions in binary format into this data type.
3. Understanding the binary formats used for the ISA and how to separate data from instructions.

One complication is a binary may be self-modifying. This means there is no *a priori* way of disassembling a binary in all cases. We ignore this complication and instead Quameleon greedily disassembles each instruction it knows.

Lifting an Instruction

Lifters to QIL are fairly straightforward thanks to Haskell's type system, the parametricity of QIL, and the simple design of (most) assembly instructions. Usually, specification of the behavior of many instructions can be accomplished in just a few lines of Haskell. As an example, consider the implementation of the logical and operation below.

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

This consists of accessing the register `ra`, a memory location `op`, performing a logical and, writing the result, setting status flags (some were elided for brevity), and finally branching to the next instruction. We note the `andBit` operation is generic in the size of the input; this allows significant code reuse and static checking that operands are well-formed.

QIL: Quameleon Intermediate Language

QIL is an intermediate language designed to capture the semantics of binary programs for a wide variety of architectures while having an easily formalized semantics.

We designed QIL from scratch so we could provide useful programming language features to ensure strong static type guarantees, provide ease of analysis, and facilitate code transformations. One interesting feature of QIL is its bit vector abstraction which provides statically typed bit vector fields and widths for any bit width. This allows greater code reuse in contrast with other ILs which have separate instructions for different bit widths.

We provide three encodings of QIL:

1. a nominal encoding useful for optimization;
2. finally tagless encoding good for code generation; and
3. de Bruijn index encoding to more easily translate representations.

We also support JSON output for our angr bridge and a pretty-printed, human-readable syntax as shown in Listing 2.

At present QIL does *not* support non-sequential semantics and self modifying programs. We discuss these limitations in the future work section.

Overview of Syntactic Elements

QIL has several fundamental type families which can be referenced by a variable

- Values: these represent bit vectors. In QIL's type system `Values` are parameterized by a natural number of bits.
- Locations: these represent assignable locations where to which values can be read and written. In QIL's type system `Locations` are parameterized by a natural number which denotes the size of `Values` storable there.
- Labels: these represent program locations and can be jumped to.
- RamSelectors: these represent families of `Locations` indexed by `Values`.
- JoinPoints: these represent a local continuation which can be jumped to. Unlike a `Label`, which denotes a global location, a `JoinPoints` type is parameterized by a list of argument types.

From these, we form instructions, blocks, and programs. A QIL program consists of four pieces of information:

1. a globally defined code-pointer size (a natural number)
2. a sequence of allocation instructions defining registers and memories
3. a sequence of blocks, each binding a label
4. optionally, a label called the "entry point."

Blocks bind labels one of two ways. Registered blocks can be used for static jumps (with an associated label) or dynamic jumps (with an associated address). Unregistered blocks only have a label.

Generally, in a lifter a registered block is (at least initially) generated for each instruction. Optimization may then generate additional blocks or combine blocks via inlining.

In either kind of block, the body of the block consists of a sequence of instructions which may bind variables. Each variable is bound exactly once in the style of *static single assignment* (SSA). Unlike many SSA ILs, there are no "labels" or "ϕ nodes" inside a QIL block. Instead, block-local control is achieved by way of structured control flow consisting of:

- boolean (if) and multiway (case) conditional statements
- let-bound join points (which take parameters such as functions in high level languages).

Overview of Semantics

A QIL program takes as its denotation a labeled transition system where the labels on transitions are sequences of well-typed, reads and writes to some set of locations.

As is standard, we think of the abstract state of the program's denotation as coming from two parts: program locations (that is, the QIL labels bound by blocks) and the other state, the latter of which is described by the variables (RAM, registers, etc.) defined in the allocation section of the QIL program.

By computing the denotation of each block body, we can easily compute the denotation of the entire program. Specifically, we start with, as states, the Cartesian product of the set of labels and the denotation of the domain of memory (both RAM and registers), and then for every element of the denotation of a block we add the appropriate transitions, adding intermediate steps as necessary.

Note that, crucially, while QIL blocks can be nondeterministic, they must terminate. No fancy denotational techniques are needed to account for non-termination, as it exists only at the top-level of the semantics (the transition system).

A Worked Example Program

In order to demonstrate the QIL language, consider the fragment of M6800 assembly language in Listing 1 which takes the bitwise and of 0xF and the byte at location 0x40.

Listing 1. A fragment of M6800 assembly.

```
LDA A #15  ; A <- 0xF
AND A $40  ; A <- A & [0x40]
```

Listing 2. A worked QIL example.

```
1  code_ptr_size: S16
2  alloc_part: {
3     &1 := alloc[S8] // Register A
4     // ... Other registers
5     &6 := alloc[S1] // Carry Flag
6     // ... Other status flags
7     MEM(1) := buildMemory[S16 S8]
8  }
9  code_part: {
10    @1 := block{ }
11    @2 := registered_block "AND A (DIR8 64)" 2 {
12       %1 := readLoc[S8] &1 // read Register A
13       &12 := MEM(1)[S16].BV[S8](40)
14       %2 := readLoc[S8] &12
15       %3 := AndBit[S8] %1 %2
16       writeLoc[S8] &1 %3 // set Register A
17       branch @1
18    }
19    @3 := registered_block "LDA A (IMM8 15)" 0 {
20       writeLoc[S8] &1 BV[S8](f) // set Register A
21       branch @2
22    }
23    @4 := block{
24       branch @3
25    }
26 }
27 entry_point: @4
```

The pretty-printed QIL is given in Listing 2. Note that this program is unoptimized: after sufficient optimization this program would reduce to a precomputed write since the program has no inputs.

Line 1 indicates this program uses 16-bit values for dynamic jumps. Lines 2–8 set up the memory used by this machine. For instance, Line 3 creates an 8-bit assignable memory location (i.e. a register) and associates it with the name `&1`. In the QIL syntax all location variables start with an ampersand and the comment `Register A` is just metadata.

Our M6800 lifter ends up generating blocks in the opposite order from the instructions. As such, the initial instruction, set as the entry point, has the label `@4`. Next, the program branches to the label `@3`. Line 20 states the location `&1` gets the value 15 (0xF), and its size is 8 bits. The other potentially confusing line is 13, which reads 8 bits from the 16-bit memory location 64 (hex 0x40).

Optimizations

Quameleon provides several optimization passes with the goal of decreasing code size and increasing analyzability. One example is constant folding, which consists of replacing a variable with its value when that value can be statically

known. Other optimizations we have implemented include unreachable code elimination and inlining.

Analysis Backends

We have currently implemented two analysis backends.

1. A bridge between Quameleon and angr. This allows loading QIL programs so that QIL appears as simply another binary format. We include metadata such as the register names inside this JSON to provide similar convenience to existing ISAs.
2. Concrete execution. This backend provides the ability to interpret QIL programs; i.e. an emulator for supported architectures. Our general purpose interpreter takes a set of call-backs for resolving I/O effects, early termination, or non-deterministic calls. As such, we provide a unified backend for both pure and side effectful interpretation strategies.

Related Work

Related work includes analysis tools such as BAP (Binary Analysis Platform) [1], B2R2 [3], and angr [6].

In particular, angr has a large user community and a substantial degree of completeness. Unfortunately, neither angr nor BAP supported the ISAs we needed. We note the differing design goals of other tools to motivate the overall design of Quameleon.

- Our goal is to generalize both frontends and backends for binaries which we know at some level the expected behavior, but require high assurance of correctness; this contrasts with angr's design goals of being primarily for reverse engingeering adversarial binaries using symbolic execution and heuristics.
- Both angr and BAP use ILs based on mutable temporaries by default. Instead, we wanted a static single assignment (SSA)-based IL. LLVM [4] uses SSA, but is aimed for optimization rather than binary analysis.

Future Work

The first desired feature would be to support self-modifying binaries. Our idea to the end is to extend QIL with an (optional) special block handling branching to values not known until runtime, wherein QIL could look up the location in memory, decode its contents to an instruction, then evaluate that instruction.

We also wish to add additional backends as listed in Fig. 1 such as Hoare Logic-style predicate transformers and abstract interpretation.

Lastly, QIL does not include floating point instructions unlike many other ILs. We are exploring what it would take to develop a formal theory of all floating point representations to be used in QIL that would be generic enough for pre-IEEE-754 floating point formats.

Conclusion

We presented Quameleon, a tool for sound binary analysis designed from the beginning to be easily extended to different architectures and types of analysis. We accomplish this by lifting our input ISAs into an intermediate language QIL, an SSA-based intermediate language. QIL programs are amenable to analysis because they make explicit all effects of an assembly language program and the small core language facilitates our effort to formalize QIL in a proof assistant such as Coq.

Another result of extensibility being a primary design goal is the ability to extend to old ISAs; languages with non-byte addressable memory, pre-IEEE-754 floating point arithmetic, or requiring cycle-accurate emulation can all be added without modification QIL's core.

Acknowledgments

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.

References

[1] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. Bap: A binary analysis platform. In *Computer Aided Verification (CAV)* (Snowbird, UT, USA, July 2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *LNCS*, Springer Berlin Heidelberg, pp. 463–469.

[2] Hex-Rays. The ida disassembler and debugger, 2018. Available at https://www.hex-rays.com.

[3] Jung, M., Kim, S., Han, H., Choi, J., and Cha, S. K. B2R2: Building an efficient front-end for binary analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research* (2019).

[4] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, CA, USA, Mar. 2004), CGO '04, IEEE Computer Society, pp. 75–.

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

[6] 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.