The BAP Handbook

David Brumley, Ivan Jager, Edward J. Schwartz, and Spencer Whitman

April 10, 2014
# Contents

1 Introduction 1
   1.1 The Need for Binary Analysis 1
   1.2 Desired Properties 4
   1.3 BAP Overview 5

2 The BAP Toolkit 9
   2.1 Toolkit Overview 9
   2.2 Obtaining BAP 9
   2.3 **toil**: Lifting Binary Code to BIL 9
   2.4 **iltrans**: Programs Transformations on BIL Code 11
   2.5 **topredicate**: Predicate-based Program Verification 13
   2.6 **ileval**: Concrete evaluation 13
   2.7 **codegen**: LLVM-based code generation 13
   2.8 Discussion 13
      2.8.1 Why Design a New Infrastructure? 13
      2.8.2 Limitations of BAP 14
      2.8.3 Is Lifting Correct? 15
      2.8.4 Size of Lifting IL for a Program 15

3 Formalization of BIL 17
   3.1 The BIL Language 17
      3.1.1 Values and Types 17
      3.1.2 Expressions 18
      3.1.3 Statements and Programs 19
   3.2 Normalized Memory 20
      3.2.1 Operational Semantics 21

4 BAP Development Guide 25
   4.1 Overview 25
CONTENTS

5 Examples 27
  5.1 Generating and Working with IL 27
    5.1.1 Lifting to the IL 27
    5.1.2 Built-in Graphs and Analyses 30
    5.1.3 Verification Conditions 36
  5.2 Functions 37
  5.3 Concrete Evaluation 38
  5.4 Traces 39
    5.4.1 Setup 40
    5.4.2 Recording a trace 41

6 Related Work 43

7 Credits 45
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A three instruction x86 program, and its corresponding control flow graph. Note the logic for deciding when the jump on overflow instruction (statement 3) is taken, when is often omitted by other binary analysis platforms.</td>
</tr>
<tr>
<td>1.2</td>
<td>The semantics of the <code>rep</code> instruction according to Intel [15]. Modern architectures often have hundreds of instructions that have complex semantics like <code>rep</code>.</td>
</tr>
<tr>
<td>1.3</td>
<td>The BAP binary analysis architecture and components. BAP is divided into a front-end, which is responsible for lifting instructions to the BAP IL, and a platform-independent back-end for analyses.</td>
</tr>
<tr>
<td>2.1</td>
<td>High-level view of tools provided with BAP.</td>
</tr>
<tr>
<td>3.1</td>
<td>(a) shows an example of little-endian stores as found in x86 that partially overlap. (b) shows memory after executing line 1, and (c) shows memory after executing line 5. Line 7 will load the value 0x22bb.</td>
</tr>
<tr>
<td>3.2</td>
<td>Normalized version of the store and load from Figure 3.1a.</td>
</tr>
<tr>
<td>5.1</td>
<td>Example Basic CFG</td>
</tr>
<tr>
<td>5.2</td>
<td>Example Detailed CFG</td>
</tr>
<tr>
<td>5.3</td>
<td>Example CDG</td>
</tr>
<tr>
<td>5.4</td>
<td>Example DDG</td>
</tr>
<tr>
<td>5.5</td>
<td>Example Simplified CFG</td>
</tr>
</tbody>
</table>
List of Tables

3.1 The Binary Intermediate Language. Note commas separate operators. ........................................ 18
3.2 Operational semantics notation. ........................................ 22
3.3 Operational Semantics. ........................................ 23
Chapter 1

Introduction

1.1 The Need for Binary Analysis

Binary code is everywhere. In most situations, users only have access to code in binary (i.e., executable) form. Most common, off-the-shelf (COTS) software (e.g., Microsoft Windows, Adobe Acrobat, etc.) is only available to end-users in binary form. Malicious code (i.e., malware) created by attackers is typically only available in binary form. The ubiquity of binary code ensures that security techniques which only require access to the program binary are likely to be widely applicable. Further, binary code analysis allows us to argue about the security of the code that will run, not just the code that was compiled.

A binary-centric approach to software security requires the ability to perform program analysis on binary code. A program analysis (whether it be static or dynamic) is an algorithm for determining the effects of a set of statements in the programming language under consideration. Thus, a binary-centric approach requires 1) the ability to analyze each instruction in a manner faithful to its semantics, and 2) a method for encoding an algorithm over those instructions.

However, there are two primary challenges to performing accurate and faithful analysis on modern binary code. First, code analysis at the binary level is different than source code because binary code lacks many abstractions found in higher-level languages. Second, binary code is significantly more complex than source code, which introduces new engineering challenges for binary code analysis.
CHAPTER 1. INTRODUCTION

Binary Code Analysis is Different than Source Code Analysis. Binary code is different than source code. Thus, we must develop and only use program analyses that are suitable for the unique characteristics of binary code. In particular, binary code lacks abstractions that are often fundamental to source code and source code analysis, such as:

- **Functions.** The function abstraction does not exist at the binary level. Instead, control flow in a binary program is performed by jumps. For example, the x86 instruction `call x` is just syntactic sugar (i.e., shorthand) for storing the number in the instruction pointer register `eip` at the address named by the register `esp`, decrementing `esp` by the architecture word size, and then loading the `eip` with number `x`. Indeed, it is perfectly valid in assembly, and sometimes happens in practice, that one may call into the middle of a “function”, or have a single “function” separated into non-contiguous pieces. The lack of a function abstraction poses significant scalability challenges to binary analysis.

- **Memory vs. Buffers.** Binary code does not have buffers, it has memory. While the OS may determine a particular memory page is not valid, memory does not have the traditional semantics of a user-specified type and size. One implication of the difference between buffers and memory is that in binary code there is no such thing as a buffer overflow. While we may say a particular store violates a higher-level semantics given by the source code, such facts are inferences with respect to the higher-level semantics, not part of the binary code itself. The lack of buffers means we have to conservatively reason about the entire memory space (unless proven otherwise) at each operation.

- **No Types.** New types cannot be created or used since there is no such thing as a type constructor in binary code. The only types available are those provided by the hardware: registers and memory. Even register types are not necessarily a good choice, since it is common to store values from one register type (e.g., 32-bit register) and read them as another (e.g., 8-bit register). The lack of types means we cannot use type-based analysis, which is often instrumental in scaling traditional analyses.

Binary Code is Complex. One approach is to disassemble binary code into a sequence of assembly instructions, and then perform program analysis
1.1. THE NEED FOR BINARY ANALYSIS

// instr dst, src
1. add eax, ebx
2. shl eax, edx
3. jo target
4. ....
...
target: ...

Figure 1.1: A three instruction x86 program, and its corresponding control flow graph. Note the logic for deciding when the jump on overflow instruction (statement 3) is taken, when is often omitted by other binary analysis platforms.
directly over the assembly instructions. This type of assembly-specific approach is naïve because each analysis would have to individually understand the semantics of the assembly, which is difficult.

For example, consider the basic program analysis problem of determining when the conditional jump is taken on line 3 of Figure 1.1. In this example, all operands are 32-bit registers, and all arithmetic is therefore performed mod \(2^{32}\). Instruction 1 computes the sum \(\text{eax} := \text{eax} + \text{ebx}\) (mod \(2^{32}\)). Instruction 2 computes \(\text{eax} = \text{eax} \ll \text{edx}\). Both instructions may set the overflow status register \(\text{OF}\). The \texttt{add} instruction will set the \(\text{OF}\) flag if \(\text{eax} + \text{ebx} \geq 2^{32}\). The \texttt{shl} instruction will set the \(\text{OF}\) flag if \(\text{edx}\) is 1 and the top two bits of \(\text{eax}\) are not equal on line 2. The instruction on line 3 tests to see if \(\text{OF}\) is set, and if so, jumps to \texttt{target}, else executes the next sequential instruction.

In order to determine when the jump on line 3 is taken, an analysis must reconstruct all side-effects of \texttt{add} and \texttt{shl}. The proper control flow diagram is shown in Figure 1.1. As demonstrated by this three line program, even though a program may look simple, determining the effects may be complicated. An assembly-specific approach would require each analysis to reason about such complex semantics.

Worse, many architectures allow instruction prefixes, which can further complicate the semantics. For example, in x86 a \texttt{rep} prefix essentially turns an instruction into a single instruction loop, e.g., \texttt{rep i s, d} will repeatedly execute operation \(i\) on arguments \(s\) and \(d\). The exact semantics (per Intel [15]) of \texttt{rep} are shown in Figure 1.2. An assembly-specific approach would have to consider this complicated logic for any instruction that may carry the \texttt{rep} prefix. If analyses are written directly on assembly, each analysis would duplicate this logic.

If there were only a few instructions, perhaps the complexity would not be too onerous. Modern architectures, however, typically have hundreds of instructions. x86, for example, has well over 300 instructions (which are documented in over 11 lbs of manuals [15]).

### 1.2 Desired Properties

We would like a platform for analyzing binary code that 1) supports writing analyses in a concise and straightforward fashion, 2) provides abstractions for common semantics across all assemblies, and 3) is architecture independent when possible.

We want an architecture that supports writing analyses in a concise,
1.3 BAP Overview

BAP is designed to facilitate faithful security-relevant binary program analysis by 1) reducing complex instruction sets to a single, small, and formally speci-
Figure 1.3: The BAP binary analysis architecture and components. BAP is divided into a front-end, which is responsible for lifting instructions to the BAP IL, and a platform-independent back-end for analyses.
fied intermediate language that supports writing concise, easy-to-understand analyses 2) providing a set of core program analyses abstractions, and 3) being architecture independent when possible in order to support easy re-targeting.

Figure 1.3 shows a high-level picture of BAP. BAP is divided into an architecture-specific front-end and an architecture-independent back-end. At the core of BAP is an architecture-independent intermediate language (IL), called BIL, for assembly. Assembly instructions in the underlying architecture are lifted up to the BIL via the BAP front-end. All analyses are performed on the platform-independent BIL in the back-end.

We lift to the BIL by using open-source utilities to parse the binary format and produce assembly. The assembly is then lifted up to the BAP IL in a syntax-directed manner. The BAP front-end currently supports lifting usermode x86 [15] code, though other architectures may be added to the BAP framework.

The BAP back-end supports a variety of core program analyses and utilities. The back-end has utilities for creating a variety of different graphs, such as control flow and program dependence graphs. The back-end also provides an optimization framework. The optimization framework is usually used to simplify a specific set of instructions. We also provide program verification capabilities such as symbolic execution, calculating the weakest precondition, and interfacing with decision procedures. BAP can also write out lifted BAP instructions as valid C code via the code generator back-end.
Chapter 2

The BAP Toolkit

2.1 Toolkit Overview

There are several tools built and distributed with BAP. First, we provide a tool called toil which lifts binary code to BIL. Second, we provide a tool called iltrans which takes as input BIL code, transforms it, and then outputs the resulting BIL code. Thus, iltrans is a mapping from BIL to BIL. Finally, we provide a set of additional utilities such as one that evaluates BIL programs, and another that calculates the weakest precondition for a BIL program. Figure 2.1 shows the relationship of these tools. Different output options include optimizing the IL, generating CFG’s, PDG’s, DDG’s, and CDG’s, calculating the weakest precondition, and several other options. We describe these tools at a high level in this chapter.

Note: Each tool provides a -help option, which provides the latest set of command-line options.

2.2 Obtaining BAP

BAP can be found at http://bap.ece.cmu.edu

2.3 toil: Lifting Binary Code to BIL

The toil tool lifts binary code to BIL. Lifting up binary code to the IL consists of two steps:

- First, the binary file is opened using libbfd. BAP can read binaries from any bfd-supported binary, including ELF and PE binaries.
• Second, the executable code in the binary is disassembled. \texttt{toil} currently uses a linear sweep disassembler.

• Each assembly instruction discovered by the linear sweep disassembler is then lifted directly to BIL.

Lifted assembly instructions have all of the side-effects explicitly exposed. As a result, a single typical assembly instruction will be lifted as a sequence of BIL instructions. For example, the \texttt{add} \$2, \%eax instruction is lifted as:

\begin{verbatim}
extrue x = 0x2
label pc_0x0
T_t1:u32 = R_EAX:u32
T_t2:u32 = 2:u32
R_EAX:u32 = R_EAX:u32 + T_t2:u32
R_CF:bool = R_EAX:u32 < T_t1:u32
R_OF:bool = high:bool((T_t1:u32 ^ ~T_t2:u32) & (T_t1:u32 ^ R_EAX:u32))
R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EAX:u32 ^ T_t1:u32 ^ T_t2:u32))
R_EAX:u32 >> 1:u32 ^ R_EAX:u32)
R_SF:bool = high:bool(R_EAX:u32)
R_ZF:bool = 0:u32 == R_EAX:u32
\end{verbatim}
The lifted BIL code explicitly detail all the side-effects of the \texttt{add} instruction, including all six flags that are updated by the operation. As another example, an instruction with the \texttt{rep} prefix (whose semantics are in Figure 1.2) is lifted as a sequence of instructions that form a loop.

In addition to binary files, \texttt{toil} can also lift an instruction trace to the IL. The most recent BAP trace format can be lifted using the \texttt{-serializedtrace} option.

\texttt{toil} can output to several formats for easy parsing, including protobuf, JSON, and XML. These formats are selected via the \texttt{-topb}, \texttt{-tojson}, and \texttt{-toxml} options. Code for reading the protobuf encoding is in the \texttt{piqi-files/protobuf} directory.

### 2.4 iltrans: Programs Transformations on BIL Code

The \texttt{iltrans} tool applies transformations and analyses to BIL. The arguments to \texttt{iltrans} specify a sequence of transformations, optionally printing out the result of the transformation. For example, executing:

\begin{verbatim}
iltrans -trans1 -trans2 outfile -trans3 infile
\end{verbatim}

would read in \texttt{infile}, perform transformation \texttt{trans1}, the output of which is fed to \texttt{trans2}, the output of which is again fed to \texttt{trans3} as well as printed to \texttt{outfile}. Although this may seem a little strange at first, the tool is designed to make it easy to specify exactly what algorithms you wish to perform over the IL.

\texttt{iltrans} provides a library of common analyses and utilities, which can be used as building blocks for more advanced analyses. Please consult \texttt{iltrans -help} for exact arguments. We detail below at a high level several of the analysis that can be performed by \texttt{iltrans}

**Graphs.** \texttt{iltrans} provides options for building and manipulating control flow graphs (CFG), including a pretty-printer for the graphviz DOT graph language \texttt{[2]}. \texttt{iltrans} also provides options for building data, control, and program dependence graphs \texttt{[22]}. These graphs can be output in the \texttt{dot} language. The graphs themselves can be quite large. We currently use \texttt{zgrviewer} \texttt{[4]} for viewing the resulting dot files.

One issue when constructing graphs of an assembly program is determining the successors of jumps to computed values, called \textit{indirect} jumps. Resolving indirect jumps usually involves a program analysis that require a
CFG, e.g., VSA [6]. Thus, there is a potential circular dependency. Note that an indirect jump may potentially go anywhere, including the heap or code that has not been previously disassembled.

Our solution is to designate a special node as a successor of unresolved indirect jump targets in the CFG. We provide this so an analysis that depends on a correct CFG can recognize that we do not know the subsequent state. For example, a data-flow analysis could widen all facts to the lattice bottom. Most normal analyses will first run an indirect jump resolution analysis in order to build a more precise CFG that resolves indirect jumps to a list of possible jump targets.

**Single Static Assignment.** \texttt{iltrans} supports conversion to and from single static assignment (SSA) form [22]. SSA form makes writing an analysis easier because every variable is defined statically only once. Note we convert both memory and scalars to SSA form. We convert memories so that an analysis can syntactically distinguish between memories before and after a write operation instead of requiring the analysis itself to maintain similar bookkeeping. For example, in the memory normalization example in Figure 3.2, an analysis can syntactically distinguish between the memory state before the write on line 1, the write on line 5, and the read on line 7.

**Chopping.** Given a source and sink node, a program chop [16] is a graph showing the statements that cause definitions of the source to affect uses of the sink. For example, chopping can be used to restrict subsequent analyses to only a portion of code relevant to a given source and sink instead of the whole program.

**Data-flow and Optimizations.** \texttt{iltrans} interfaces with the generic data-flow engine in BAP. The data-flow engine works on user-defined lattices, such as those for constant propagation, dead code elimination, etc. BAP currently implements Simpson’s global value numbering [29], (aggressive) dead-code elimination [22], and live-variable analysis [22].

We have also implemented value set analysis [6]. Value set analysis is a data-flow analysis that over-approximates the values for each variable at each program point. Value-set analysis can be used to help resolve indirect jumps. It can also be used as an alias analysis. Two memory accesses are potentially aliased if the intersection of their value sets is non-empty.

Optimizations are useful for simplifying or speeding up subsequent analyses. For example, we have found that the time for the decision procedure
2.5. **toppredicate: Predicate-based Program Verification**

STP to return a satisfying answer for a query can be cut in half by first using program optimization to simplify the query [9].

**toppredicate** supports program verification in two ways. First, **toppredicate** can convert the IL into Dijkstra’s Guarded Command Language (GCL), and calculate the weakest precondition with respect to GCL programs [13]. The weakest precondition for a program with respect to a predicate \( q \) is the most general condition such that any input satisfying the condition is guaranteed to terminate (normally) in a state satisfying \( q \). Currently we only support acyclic programs, i.e., we do not support GCL while.

**toppredicate** also interfaces with satisfiability modulo theory (SMT) solvers. **toppredicate** can write out expressions (e.g., weakest preconditions) in the CVC Lite [1], SMTLIB1 [28], and SMTLIB2 [8] formats, one of which is supported by virtually all solvers.

2.6 **ileval: Concrete evaluation**

**ileval** evaluates a given BIL program using the operational semantics in in 3.2.1. The evaluator allows us to execute programs without recompiling the IL back down to assembly. For example, we can test that a raised program is correct by executing the IL on an input \( i \), observing the value \( v \), executing the original binary program on \( i \), observing the value \( v' \), and verifying \( v = v' \).

2.7 **codegen: LLVM-based code generation**

codegen converts a BIL program to the LLVM [18] IL, which can then be converted to native code using LLVM’s code generators. This may be useful for translating binary code to other architectures, or for static instrumentation.
CHAPTER 2. THE BAP TOOLKIT

2.8 Discussion

2.8.1 Why Design a New Infrastructure?

At a high level, we designed BAP as a new platform because existing platforms are a) defined for higher-level languages and thus not appropriate for binary code analysis, b) designed for orthogonal goals such as binary instrumentation or decompilation, and/or c) unavailable to use for research purposes. As a result, other tools we tried (e.g., DynInst [27] version 5.2, Phoenix [20] April 2008 SDK, IDA Pro [12] version 5, and others) were inadequate, e.g., could not create a correct control flow graph for the 3 line assembly shown in Figure 1.1. Another reason we created BAP is so that we could be sure of the semantics of analysis. Existing platforms tend not to have publicly available formally defined semantics, thus we would not know exactly what we are using.

Formal semantics are important for several practical reasons. We found that defining the semantics of BAP was helpful in catching design bugs, unsupported assumptions, and other errors that could affect many different kinds of analysis. In addition, the semantics are helpful for communicating how BAP works with other researchers. Further, without a specified semantics, it is difficult to show an analysis is correct. For example, in [10] we show our proposed assembly-level alias analysis [10] is correct with respect to the operational semantics of BAP.

2.8.2 Limitations of BAP

Architectures BAP only supports x86 right now. We intend to add support for x86-64 and ARM in the future.

Semantics BAP is designed to enable program analysis of binary program states. Therefore, analyses that depend upon more than the operational semantics of the instruction fall outside the scope of BAP. For example, creating an analysis of the timing behavior of binary programs falls outside the current scope of BAP.

User-mode instructions BAP only models user-mode level instructions. Unhandled instructions will be lifted as special statements or unknown expressions.

Integer instructions BAP only models instructions that manipulate integers. In particular, BAP does not model floating point instructions.
Indirect jumps BAP does not currently resolve indirect jumps, or perform “CFG recovery”. We plan to add this in a future release.

2.8.3 Is Lifting Correct?

Assembly instructions are lifted to BAP in a syntax directed manner. One may view the BAP IL as a model of the underlying assembly. There is a chance, however, that lifting could produce incorrect IL. Although it is impossible to say that all assembly instructions are correctly lifted, the advantage of BAP’s design is that only the lifting process needs to understand the semantics of the original assembly.

We perform nightly testing to make sure that BAP’s model of execution matches what happens on a real x86 processor. Our nightly tests also tell us which instructions are used in our test programs that are not modeled in BAP. The results of these tests are always available at http://bap.ece.cmu.edu/nightly-reports/.

2.8.4 Size of Lifting IL for a Program

A single assembly instruction will typically be translated into several BAP instructions. Thus, the resulting BAP program will have more statements than the corresponding assembly. For example and roughly speaking, x86 assembly instructions are raised to be about 7 BAP instructions: 1 BAP instruction for the direct effect, and 6 for updating processor-specific status flags.

In our experience, the constant-size factor in code size is worth the benefits of simplifying the semantics of assembly. Assembly instructions are designed to be efficient for a computer to execute, not for a human to understand. The IL, on the other hand, is designed to be easy for a human to understand. We have found even experienced assembly-level programmers will comment that they have a hard time keeping track of control and data dependencies since there are few syntactic cues in assembly to help. BAP, on the other hand, obviates all data and control dependencies within the code.
Chapter 3

Formalization of BIL

At the core of BAP is the BAP intermediate language, called BIL. In this chapter we present a formalization of BIL. The formalization is intended to provide an exact description of what we mean by statements in the IL.

Note: In our implementation, as discussed elsewhere, there are actually several IL’s. Throughout this chapter we discuss the IL as implemented in ast.ml. The other IL’s differ in uninteresting ways and are variants of the IL discussed here. In particular, developers will be interested in ssa.ml, which defines the IL in SSA form, and also does not allow recursively defined expressions (i.e., it is an SSA three-address code). Thus, this chapter can be viewed as a specification for the actual code.

3.1 The BIL Language

Table 3.1 shows the syntax of BIL. We use the term “instruction” to refer to an assembly-level instruction, and the term “statement” to refer to instructions within BIL. Thus, BAP raises instructions to statements in BIL. In 3.2.1 we provide the operational semantics. In the remainder of this section we give an informal description and motivation for constructs in the IL.

3.1.1 Values and Types

The base types $\tau_{\text{reg}}$ in BIL IL are 1, 8, 16, 32, and 64-bit registers (i.e., $n$-bit vectors), and memories. Memories are given type $\text{mem}_t(\tau_{\text{reg}})$, where $\tau_{\text{reg}}$ determines the type for memory addresses. For example, $\text{mem}(\tau_{\text{reg32}})$ corresponds to memory on a typical 32-bit machine.

We also have arrays, which are given type $\text{array}_t(\tau_{\text{reg}}, \tau_{\text{reg}})$. The tuple
CHAPTER 3. FORMALIZATION OF BIL

program ::= stmt*
stmt ::= var := exp | jmp(exp) | cjmp(exp,exp,exp)
        | halt(exp) | assert(exp) | label label_kind | special(string)
exp ::= load(exp, exp, exp, τ_reg) | store(exp, exp, exp, exp, τ_reg) | exp ◊b exp
    | ◊u exp | var | lab(string) | integer | cast(cast_kind, τ_reg, exp)
    | let var = exp in exp | unknown(string, τ) | name(exp)
label_kind ::= integer | string
cast_kind ::= unsigned | signed | high | low
◊b ::= +, −, ∗, /, \, mod, mod_s, ≪, ≫, ≫a, &\, |, ⊕, ==, !=, =, <, ≤, s, ≤s
◊u ::= − (unary minus), ~ (bit-wise not)
value ::= integer | memory | string | ⊥
integer ::= n (τ_reg)
memory ::= { integer → integer, integer → integer, ... } (τ_mem)
τ ::= τ_reg | τ_mem
τ_mem ::= mem_t(τ_reg) | array_t(τ_reg, τ_reg)
τ_reg ::= reg1_t | reg8_t | reg16_t | reg32_t | reg64_t

Table 3.1: The Binary Intermediate Language. Note commas separate operators.

(τ_reg, τ_reg) specifies the index and element type of an array, respectively. As we will see, we use arrays to normalize endianded memory accesses.

There are three types of values in BIL. First, BIL has numbers n of type τ_reg. Second, BIL has memory values {n_{a1} → n_{v1}, n_{a2} → n_{v2}, ...}, where n_{ai} denotes a number used as an address, and n_{vi} denotes the value stored at the address. Finally, BIL has a nonsense value ⊥. ⊥ values are not exposed to the user and cannot be constructed in the presentation language. ⊥ is used internally to indicate a failed execution.

3.1.2 Expressions

Expressions in BIL are side-effect free, and are similar to those found in most languages. BIL has binary operations ◊b (note “&” and “|” are bit-wise), unary operations ◊u, constants, let bindings, and casting. Casting is used when indexing registers under different addressing modes. For example, the lower 8 bits of eax in x86 are known as al. When lifting x86 instructions, we use casting to project out the lower-bits of the corresponding eax register variable to an al register variable when al is accessed.

The semantics of load(e1, e2, e3, τ_reg) is to load from the memory speci-
3.1. THE BIL LANGUAGE

ified by $e_1$ at address $e_2$. In C, this would loosely be written $e_1[e_2]$. The parameter $e_3$ tells us the endianness to use when loading bytes from memory. $e_3$ is forced to be of type bool, where we arbitrarily affix the meaning that 0 is little endian and 1 is big endian (since 0 is “littler” than 1). Some architecture consistently use the same endianness, e.g., for x86, the value of $e_3$ will always correspond to little-endianness. However, other architectures such as ARM specify the endianness of a load at run-time. Finally, $\tau_{\text{reg}}$ tells us how many bytes to load. In C, if $e_1$ is of type $\tau$, then $e_1[e_2]$ loads $\text{sizeof}(\tau)$ bytes. $\tau_{\text{reg}}$ similarly tells us how many bytes to load from memory. While technically we could infer $\tau_{\text{reg}}$ given $e_1$, we keep it in the IL explicitly for efficiency.

In BIL, the store operation is pure (i.e., side-effect free). The advantage of pure memory operations in BIL notation is it makes it possible to syntactically distinguish what memory is modified or read. One place we take advantage of this is in SSA where both scalars and memory have a unique single static assignment location.

Each store expression must specify what memory to load or store from. The resulting memory is returned as a value. The semantics of store$(e_1, e_2, e_3, e_4, \tau_{\text{reg}})$ are to store in memory $e_1$, starting at address $e_2$ the value $e_3$. The store is performed given the endianness of $e_4$. This may seem very complicated when reading. However, the operational semantics are quite simple: you may want to read the store rules.

The last expression type of note is unknown. An unknown specifies an operation we could not lift to BIL. The purpose of an unknown is to adhere to the BAP principle to know what you do not know. Consider the case where Intel adds a new instruction, e.g., as happens in each processor revision. BAP may not know about such instructions, thus cannot raise it to BIL. One option would be to ignore such instructions. However, the result of any analysis would be suspect in this case. A sound option is to abort lifting and raise an error. However, we often end up not interested in particular instructions. Our solution is to raise such instructions, when possible, to an assignment (where the left hand side is of the correct type and name) with the actual operation left unspecified as unknown.

3.1.3 Statements and Programs

A program in BIL is a sequence of statements. There are 7 different kinds of instructions. The language has assignments, jumps, conditional jumps, and labels. The target of all jumps and conditional jumps must be a valid label in our operational semantics, else the program terminates in the error
CHAPTER 3. FORMALIZATION OF BIL

// x86 instr dst,src
1. mov [eax], 0xaabbccdd
2. mov ebx, eax
3. add ebx, 0x3
4. mov eax, 0x1122
5. mov [ebx], ax
6. sub ebx, 1
7. mov ax, [ebx]

(a) 

<table>
<thead>
<tr>
<th>address</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0xdd</td>
</tr>
<tr>
<td>eax+1</td>
<td>0xcc</td>
</tr>
<tr>
<td>eax+2</td>
<td>0xbb</td>
</tr>
<tr>
<td>eax+3</td>
<td>0xaa</td>
</tr>
<tr>
<td>eax+4</td>
<td>0x11</td>
</tr>
</tbody>
</table>

(b) 

<table>
<thead>
<tr>
<th>address</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0xdd</td>
</tr>
<tr>
<td>eax+1</td>
<td>0xcc</td>
</tr>
<tr>
<td>eax+2</td>
<td>0xbb</td>
</tr>
<tr>
<td>eax+3</td>
<td>0x22</td>
</tr>
<tr>
<td>eax+4</td>
<td>0x11</td>
</tr>
</tbody>
</table>

(c) 

Figure 3.1: (a) shows an example of little-endian stores as found in x86 that partially overlap. (b) shows memory after executing line 1, and (c) shows memory after executing line 5. Line 7 will load the value 0x22bb.

state (⊥). Note that a jump to an undefined location (e.g., a location that was not disassembled such as to dynamically generated code) results in the BAP program halting with ⊥ (see 3.2.1). A program can halt normally at any time by issuing the halt statement. We also provide assert, which acts similar to a C assert: the asserted expression must be true, else the machine halts with ⊥.

A special in BIL corresponds to a call to an externally defined procedure or function. special statements typically arise from system calls. The id of a special indexes the kind of special, e.g., what system call.

The semantics of special are up to the analysis; its operational semantics are not defined (Chapter 3.2.1). We include special as an instruction type to explicitly distinguish calls that alter the soundness of an analysis. A typical approach to dealing with special is to replace special with an analysis-specific summary function written in the BAP IL that is appropriate for the analysis.

3.2 Normalized Memory

The endianness of a machine is usually specified by the byte-ordering of the hardware. Little endian architectures such as x86 put the low-order byte first. Big-endian architecture put the high-order byte first. Some architectures, such as ARM, allow the endianness to be specified by the instruction, e.g., mov would take three arguments: the source, destination, and endianness for which to perform the move.

We must take endianness into account when analyzing memory accesses. Consider the assembly in Figure 3.1a. The mov operation on line 2 writes
3.2. NORMALIZED MEMORY

1. mem4 = let mem1 = store(mem0, eax, 0xdd, 0, reg8_t) in
   let mem2 = store(mem1, eax+1, 0xcc, 0, reg8_t) in
   let mem3 = store(mem2, eax+2, 0xbb, 0, reg8_t) in
   store(mem3, eax+3, 0xcc, 0, reg8_t);
...

5. mem6 = let mem5 = store(mem4, ebx, 0x22, 0, reg8_t) in
   store(mem5, ebx+1, 0x22, 0, reg8_t)
...

7. value = let b1 = load(mem6, ebx, 0, reg8_t) in
   let b2 = load(mem6, ebx+1, 0, reg8_t) in
   let b1' = cast(unsigned, b1, 0, reg16_t) in
   let b2' = cast(unsigned, b2, 0, reg16_t) in
   (b2' ≪ 8) | b1';

Figure 3.2: Normalized version of the store and load from Figure 3.1a.

4 bytes to memory in little endian order (since x86 is little endian). After
executing line 2, the address given by eax contains byte 0xdd, eax+1 contains
byte 0xcc, and so on, as shown in Figure 3.1b. Lines 2 and 3 set ebx =
eax+2. Line 4 and 5 write the 16-bit value 0x1122 to ebx. An analysis of
these few lines of code needs to consider that the write on line 4 overwrites
the last byte written on line 1, as shown in Figure 3.1c. Considering such
cases requires additional logic in each analysis. For example, the value
loaded on line 7 will contain one byte from each of the two stores.

We say a memory is normalized for a b-byte addressable memory if all
loads and stores are exactly b-bytes and b-byte aligned. In x86, memory is
byte addressable, so a normalized memory for x86 has all loads and stores
at the byte level. The normalized form for the write on Line 1 of Figure 3.1a
in BIL is shown in Figure 3.2. Note that the subsequent load on line 7 is
with respect to the current memory mem6.

Normalized memory makes writing program analyses involving memory
easier. Analysis is easier because normalized memory syntactically exposes
memory updates that are otherwise implicitly defined by the endianness.
As a result, analyses do not have to reason explicitly about overlapping
memory, byte order, etc. BAP provides utilities for normalizing all memory
operations so that users can write analysis more easily.

3.2.1 Operational Semantics

The operational semantics for BIL are shown in Table 3.3. The abstract
machine configuration is given by the tuple (Π, Δ, p, i) where Π is the list of
instructions, Δ is the variable context, p is the instruction pointer, and i is
CHAPTER 3. FORMALIZATION OF BIL

Contexts

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$\Pi$</th>
<th>$n \mapsto \text{instr}$</th>
<th>Maps an instruction address to an instruction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>$\Delta$</td>
<td>$id \mapsto \text{var}$</td>
<td>Maps a variable ID to its value.</td>
</tr>
<tr>
<td>Labels</td>
<td>$\Lambda$</td>
<td>$\text{label}_\text{kind} \mapsto n$</td>
<td>Maps a label to the address of the corresponding $\text{label}$ instruction number.</td>
</tr>
</tbody>
</table>

Notation

$\Delta \vdash e \Downarrow v$ Expression $e$ evaluates to value $v$ given variable context $\Delta$ as given by the expression evaluation rules.

$\Delta' = \Delta[x \leftarrow v]$ $\Delta'$ is the same as $\Delta$ except extended to map $x$ to $v$.

$\Pi \vdash p : i$ $\Pi$ maps instruction address $p$ to instruction $i$. If $p \notin \Pi$, the machine gets stuck.

$\Lambda \vdash v : p$ $\Lambda$ maps instruction label $v$ to instruction address $p$. If $v \notin \Lambda$, then machine gets stuck. In addition, a well-formed machine should have $\Pi \vdash p : i$ where $i = \text{label}_v$ $v$, otherwise the machine is stuck.

$(\Delta, p, i) \leadsto (\Delta', p', i')$ An execution step. $p$ and $p'$ are the pre and post step program counters, $i$ and $i'$ are the pre and post step instructions, and $\Delta$ and $\Delta'$ are the pre and post step variable contexts. Note $\Lambda$ and $\Pi$ are currently static, thus for brevity not included in the execution context.

Table 3.2: Operational semantics notation.
3.2. NORMALIZED MEMORY

Expressions

\[ \Delta \vdash e \downarrow v \quad \Delta' = \Delta[x \leftarrow v] \quad \Pi \vdash p + 1 : i \]

\[ \frac{\text{Assign}}{\Delta, p, x := e \Rightarrow \Delta', p + 1, i} \]

\[ \Pi[p + 1] = i \]

\[ \Delta, p, \text{label } l \Rightarrow \Delta, p + 1, i \]

\[ \frac{\text{LABEL}}{\Delta \vdash e \downarrow v} \]

\[ \Delta \vdash e \downarrow 1 \quad \Pi[p + 1] = i \]

\[ \frac{\text{ASSERT-T}}{\Delta, p, \text{assert}(e) \Rightarrow \Delta, p + 1, i} \]

\[ \Delta \vdash e \downarrow 0 \]

\[ \frac{\text{ASSERT-F}}{\Delta, p, \text{assert}(e) \Rightarrow \bot} \]

\[ \frac{\Delta \vdash e \downarrow 1 \quad \Delta \vdash e' \downarrow v' \quad \Pi \vdash p' : \text{label } v}{\Delta \vdash e \downarrow v \quad \Pi \vdash p' : \text{label } v} \]

\[ \Delta, p, \text{cast}(unsigned, \tau) \vdash e \Rightarrow \Delta', p, \text{label } v \]

\[ \frac{\text{CJMPT}}{\Delta \vdash e \downarrow v} \]

\[ \Delta \vdash e \downarrow 0 \quad \Delta \vdash e' \downarrow v \quad \Pi \vdash p' : \text{label } v \]

\[ \frac{\text{CJMPF}}{\Delta, p, \text{cast}(left, \ell_T, \ell_F) \Rightarrow \Delta, p', \text{label } v} \]

\[ \Delta \vdash e \downarrow 0 \quad \Delta \vdash e' \downarrow v \quad \Pi \vdash p' : \text{label } v \]

\[ \frac{\text{CJMPT}}{\Delta \vdash e \downarrow v} \]

\[ \Delta \vdash e \downarrow 0 \quad \Delta \vdash e' \downarrow v \quad \Pi \vdash p' : \text{label } v \]

\[ \frac{\text{CJMPF}}{\Delta, p, \text{cast}(left, \ell_T, \ell_F) \Rightarrow \Delta, p', \text{label } v} \]

No rule for special, when \( v \notin \Lambda \), and when \( p \notin \Pi \).

Table 3.3: Operational Semantics.
the current instruction. We write $\Delta' = \Delta[x \leftarrow v]$ to indicate that $\Delta'$ is the same as $\Delta$ except that variable $x$ is updated with value $v$. For simplicity, we use $\Delta$ both as a scalar and a memory context. When ambiguous, such as in the store rule, we indicate the type of the variable in $\Delta$ in parentheses. We write $\Pi[p]$ to indicate the instruction given by address $p$.

The operational semantics can be read as follows. Each step of the execution is associated with a machine configuration $M = (\Pi, \Delta, p, i)$. A transition is given by $M \leadsto M'$ where the current configuration $M$ matches the left side of $\leadsto$ in the conclusion (below the horizontal bar), resulting in a state $M'$ to the right. The transformation from $M$ to $M'$ is given by the rule premise (above the horizontal bar).

Note: BAP and BIL do not analyze dynamically generated code. Thus, in a machine state transition $(\Pi, \Delta, p, i) \rightarrow (\Pi', \Delta, p, i)$, $\Pi = \Pi'$ always. Since $\Pi$ (the list of instructions) never changes, we omit $\Pi$ from the rules for brevity. One could add support for dynamically generated code by adding rules for updating $\Pi$.

ASSIGN and LABEL are sequential instructions that carry out the respective operation, then look up and transition to the next sequential instruction $p + 1$. The semantics of LABEL is a no-op: we use labels for jump targets. ASSIGN updates the variable context $\Delta$ resulting in a new context $\Delta'$. As mentioned, there is no rule for special; any program with a special remaining may get stuck.

Control flow is handled by jmp. A jmp instruction evaluates the jump target $e$ to a value $v$, then looks up the instruction associated with $v$. The no-inst rule terminates the program in the error state when $v$ is not associated with an instruction. For example, consider the case when a program reads in user input at location $v$, then issues a jump to $v$. The user input will be decoded as instructions at run-time. However, since the instruction comes from user input, we cannot include it in the analysis. In BAP, we indicate such possibilities by terminating in error.
Chapter 4

BAP Development Guide

4.1 Overview

The BAP infrastructure is implemented in C++ and OCaml. The lifting and back-end is implemented in OCaml. The C++ portions of BAP interface with GNU BFD for parsing executable objects, and GNU libopcodes for pretty-printing the disassembly. Each disassembled instruction is lifted to the IL by OCaml code, and then passed to the back-end. We interface the C++ front-end with the OCaml back-end using OCaml via IDL generated stubs.

The BAP top-level directory structure is:

- **libasmir**: This directory contains all the code that interfaces which parses binary file formats.

- **ocaml**: This directory contains the core BAP library and routines. All the code is written in ocaml. Only core BAP code should go on this directory. If you must modify it in a project-specific way, please make a project-specific branch for those changes.

- **utils**: This contains user utilities, again written in OCaml.

- **doc**: The directory containing this documentation.
Chapter 5

Examples

In this chapter, we present some hands-on examples that may be useful to become familiar with BAP’s capabilities.

5.1 Generating and Working with IL

One of BAP’s central features is the ability to represent the semantics of binary code in a simple language called BIL (BAP Intermediate Language). This example demonstrates how to lift binary code to the IL and manipulate it in BAP. Let’s start with the following assembly file:

```assembly
add %eax, %ebx
shl %cl, %ebx
jc target
jmp elsewhere

target:
nop

elsewhere:
nop
```

5.1.1 Lifting to the IL

In this form, the program looks very simple. Let’s see what the BAP representation of the program shows us. Compile this file (basic.S) to an object file with `gcc -c basic.S -o basic.o`. We can lift this object file with
BAP's toil command: toil -bin basic.o -o basic.il. If you inspect basic.il, you should see something like:

addr 0x0 @asm "add %eax,%ebx"
label pc_0x0
t:u32 = R_EBX:u32
R_EBX:u32 = R_EBX:u32 + R_EAX:u32
R_CF:bool = R_EBX:u32 < t:u32
R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EBX:u32 ^ t:u32 ^ R_EAX:u32))
R_OF:bool = high:bool((t:u32 ^ ~R_EAX:u32) & (t:u32 ^ R_EBX:u32))
R_PF:bool =
            R_EBX:u32 >> 1:u32 ^ R_EBX:u32)
R_SF:bool = high:bool(R_EBX:u32)
R_ZF:bool = 0:u32 == R_EBX:u32
addr 0x2 @asm "shl %cl,%ebx"
lable pc_0x2
tmpDEST:u32 = R_EBX:u32
t1:u32 = R_EBX:u32 >> 0x20:u32 - (R_ECX:u32 & 0x1f:u32)
R_CF:bool =
   if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_CF:bool else low:bool(t1:u32)
R_EBX:u32 = R_EBX:u32 << (R_ECX:u32 & 0x1f:u32)
R_OF:bool =
   if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_OF:bool else
      if (R_ECX:u32 & 0x1f:u32) == 1:u32 then high:bool(R_EBX:u32) ^ R_CF:bool
      else unknown "OF <- undefined":bool
R_SF:bool =
   if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_SF:bool else high:bool(R_EBX:u32)
R_ZF:bool =
   if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_ZF:bool else 0:u32 == R_EBX:u32
R_PF:bool =
   if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_PF:bool else
               R_EBX:u32 >> 1:u32 ^ R_EBX:u32)
R_AF:bool =
   if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_AF:bool else
      unknown "AF undefined after shift":bool
addr 0x4 @asm "jb 0x0000000000000008"
You can see from this example that BAP IL is much more verbose than assembly; this is on purpose. It might make BAP IL a little more tedious to read, but it makes writing analyses much simpler. Let’s go through some of the IL line by line to explain what is happening.

```
addr 0x0 @asm "add %eax,%ebx"
```

These two statements mark the beginning of a new assembly instruction. Note the label for pc_0x0; any jump to address zero will go to this label.

```
t:u32 = R_EBX:u32
```

Here we are saving the original value of %ebx before it is modified. This original value will be used when computing flags.

```
R_EBX:u32 = R_EBX:u32 + R_EAX:u32
```

This is the primary computation of the add instruction: adding %eax to %ebx.

```
R_CF:bool = R_EBX:u32 < t:u32
```

Here we explicitly compute the value of the carry flag. After executing the add instruction, the carry flag is set if the resulting value of %ebx is less than the original value of %ebx (which is saved in t).
R_OF:bool =
    if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_OF:bool else
    if (R_ECX:u32 & 0x1f:u32) == 1:u32 then high:bool(R_EBX:u32) ^ R_CF:bool
    else unknown "OF <- undefined":bool

Here we are explicitly computing the overflow flag after the shl instruction. Note that the IL uses if then else expressions. Also note that the the overflow flag can be set to an unknown expression when the Intel semantics says the flag should be undefined.

cjmp R_CF:bool, 8:u32, "nocjmp0"

This is a conditional jump. If the condition (in this case, the carry flag) evaluates to true, control will transfer to label pc_0x8. Otherwise, it will go to the label nocjmp0.

5.1.2 Built-in Graphs and Analyses

We might want to visualize what’s going on. There are several ways to do this in BAP, including control flow graphs (CFG), control dependence graphs (CDG), and data dependence graphs (DDG). These can be generated with the iltrans tool. iltrans takes a program as input, and then applies transformations in a pipeline. For instance, we can print an overview control flow graph by coalescing and then printing the graph with iltrans -il basic.il -to-cfg -prune-cfg -coalesce-ast -pp-ast-asms out.dot. The resulting file out.dot is a file that can be processed with GraphViz’s dot command: dot -Tpdf out.dot -o out.pdf. The output should look similar to what is shown in Figure 5.1. The next visualizations will require the program to be in Single Static Assignment (SSA) form. The command iltrans -il basic.il -to-ssa will do this. If we want to print a detailed control flow graph from SSA form, we can add -pp-ssa out.dot to the end, for instance: iltrans -il basic.il -to-ssa -pp-ssa out.dot. The output should look similar to what is shown in Figure 5.2. The CDG and DDG can be generated by using -pp-ssa-cdg and -pp-ssa-ddg respectively. Examples are shown in Figures 5.3 and 5.4.

BAP also has built-in optimizations. To apply them, simply use the -ssa-simp iltrans flag. If we apply this before producing the CFG, we can see the IL is greatly simplified: iltrans -il basic.il -to-ssa -ssa-simp -pp-ssa out.dot. This is shown in Figure 5.5.
5.1. GENERATING AND WORKING WITH IL

Figure 5.1: Example Basic CFG
CHAPTER 5. EXAMPLES

```
BB_Entry
/* entry node */
BB_0
addr 0x0 @asm "add    %eax,%ebx"
label pc_0x0

  t:u32 = R_EBX:u32
  R_EBX_74:u32 = R_EBX:u32 + R_EAX:u32
  R_CF:bool = R_EBX_74:u32 < t:u32
  temp:u32 = R_EBX_74:u32 ^ t:u32
  temp_77:u32 = temp:u32 ^ R_EAX:u32
  temp_78:u32 = 0x10:u32 & temp_77:u32
  R_AF:bool = 0x10:u32 == temp_78:u32
  temp_80:u32 = ~R_EAX:u32
  temp_81:u32 = t:u32 ^ temp_80:u32
  temp_82:u32 = t:u32 ^ R_EBX_74:u32
  temp_83:u32 = temp_81:u32 & temp_82:u32
  R_OF:bool = high:bool(temp_83:u32)
  temp_85:u32 = R_EBX_74:u32 >> 7:u32
  temp_86:u32 = R_EBX_74:u32 >> 6:u32
  temp_87:u32 = temp_85:u32 ^ temp_86:u32
  temp_88:u32 = R_EBX_74:u32 >> 5:u32
  temp_89:u32 = temp_87:u32 ^ temp_88:u32
  temp_90:u32 = R_EBX_74:u32 >> 4:u32
  temp_91:u32 = temp_89:u32 ^ temp_90:u32
  temp_92:u32 = R_EBX_74:u32 >> 3:u32
  temp_93:u32 = temp_91:u32 ^ temp_92:u32
  temp_94:u32 = R_EBX_74:u32 >> 2:u32
  temp_95:u32 = temp_93:u32 ^ temp_94:u32
  temp_96:u32 = R_EBX_74:u32 >> 1:u32
  temp_97:u32 = temp_95:u32 ^ temp_96:u32
  temp_98:u32 = temp_97:u32 ^ R_EBX_74:u32
  temp_99:bool = low:bool(temp_98:u32)
  R_PF:bool = ~temp_99:bool
  R_SF:bool = high:bool(R_EBX_74:u32)
  R_ZF:bool = 0:u32 == R_EBX_74:u32

BB_Exit
/* exit node */
BB_1
addr 0x2 @asm "shl    %cl,%ebx"
label pc_0x2

  tmpDEST:u32 = R_EBX:u32
  temp:u32 = R_ECX:u32 & 0x1f:u32
  temp_105:u32 = 0x20:u32 - temp:u32
  t1:u32 = R_EBX:u32 >> temp_105:u32
  temp_107:u32 = R_ECX:u32 & 0x1f:u32
  temp_108:bool = temp_107:u32 == 0:u32
  temp_109:bool = low:bool(t1:u32)
  temp_111:u32 = R_ECX:u32 & 0x1f:u32
  R_EBX_112:u32 = R_EBX:u32 << temp_111:u32
  temp_113:u32 = R_ECX:u32 & 0x1f:u32
  temp_114:bool = temp_113:u32 == 0:u32
  temp_115:u32 = R_ECX:u32 & 0x1f:u32
  temp_116:bool = temp_115:u32 == 1:u32
  temp_117:bool = high:bool(R_EBX_112:u32)
  temp_118:bool = temp_117:bool ^ R_CF:bool
  temp_119:bool = unknown "OF <- undefined":bool
  temp_120:bool = if temp_116:bool then temp_118:bool else temp_119:bool
  R_OF:bool = if temp_114:bool then R_OF_84:bool else temp_120:bool
  temp_122:u32 = R_ECX:u32 & 0x1f:u32
  temp_123:bool = temp_122:u32 == 0:u32
  temp_124:bool = high:bool(R_EBX_112:u32)
  R_SF:bool = if temp_123:bool then R_SF_101:bool else temp_124:bool
  temp_126:u32 = R_ECX:u32 & 0x1f:u32
  temp_127:bool = temp_126:u32 == 0:u32
  temp_128:bool = 0:u32 == R_EBX_112:u32
  R_ZF:bool = if temp_127:bool then R_ZF_102:bool else temp_128:bool
  temp_130:u32 = R_ECX:u32 & 0x1f:u32
  temp_131:bool = temp_130:u32 == 0:u32
  temp_132:u32 = R_EBX_112:u32 >> 7:u32
  temp_133:u32 = R_EBX_112:u32 >> 6:u32
  temp_134:u32 = temp_132:u32 ^ temp_133:u32
  temp_135:u32 = R_EBX_112:u32 >> 5:u32
  temp_136:u32 = temp_134:u32 ^ temp_135:u32
  temp_137:u32 = R_EBX_112:u32 >> 4:u32
  temp_138:u32 = temp_136:u32 ^ temp_137:u32
  temp_139:u32 = R_EBX_112:u32 >> 3:u32
  temp_140:u32 = temp_138:u32 ^ temp_139:u32
  temp_141:u32 = R_EBX_112:u32 >> 2:u32
  temp_142:u32 = temp_140:u32 ^ temp_141:u32
  temp_143:u32 = R_EBX_112:u32 >> 1:u32
  temp_144:u32 = temp_142:u32 ^ temp_143:u32
  temp_145:u32 = temp_144:u32 ^ R_EBX_112:u32
  temp_146:bool = low:bool(temp_145:u32)
  temp_147:bool = ~temp_146:bool
  R_PF:bool = if temp_131:bool then R_PF_100:bool else temp_147:bool
  temp_149:u32 = R_ECX:u32 & 0x1f:u32
  temp_150:bool = temp_149:u32 == 0:u32
  temp_151:bool = unknown "AF undefined after shift":bool
  R_AF:bool = if temp_150:bool then R_AF_79:bool else temp_151:bool

BB_2
addr 0x4 @asm "jb     0x0000000000000008"
label pc_0x4
cjmp R_CF:bool, 8:u32, "nocjmp0"

BB_3
label nocjmp0
addr 0x6 @asm "jmp    0x0000000000000009"
label pc_0x6
jmp 9:u32

BB_4
addr 0x8 @asm "nop" label pc_0x8
addr 0x9 @asm "nop" label pc_0x9
```

Figure 5.2: Example Detailed CFG
Figure 5.3: Example CDG
CHAPTER 5. EXAMPLES

Figure 5.4: Example DDG
5.1. GENERATING AND WORKING WITH IL

Figure 5.5: Example Simplified CFG
5.1.3 Verification Conditions

Let’s say we want to know if we can take the jump at address four. We can test this in BAP by using verification conditions (VCs). To do this, we need to add a helper variable to the IL that represents success. We can do this by setting the goal variable to false at the beginning of the program, and setting it to true after the branch is taken. Below is the modified IL that does this.

goal:bool = false
addr 0x0 @asm "add %eax,%ebx"
lbl pc_0x0

\begin{verbatim}
t:u32 = R_EBX:u32
R_EBX:u32 = R_EBX:u32 + R_EAX:u32
R_CF:bool = R_EBX:u32 < t:u32
R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EBX:u32 ^ t:u32 ^ R_EAX:u32))
R_OF:bool = high:bool((t:u32 ^ ~R_EAX:u32) & (t:u32 ^ R_EBX:u32))
R_PF:bool =
R_SF:bool = high:bool(R_EBX:u32)
R_ZF:bool = 0:u32 == R_EBX:u32
\end{verbatim}

addr 0x2 @asm "shl %cl,%ebx"
lbl pc_0x2
tmpDEST:u32 = R_EBX:u32
t1:u32 = R_EBX:u32 >> 0x20:u32 - (R_ECX:u32 & 0x1f:u32)
R_CF:bool =
  if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_CF:bool else low:bool(t1:u32)
R_EBX:u32 = R_EBX:u32 << (R_ECX:u32 & 0x1f:u32)
R_OF:bool =
  if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_OF:bool else
    if (R_ECX:u32 & 0x1f:u32) == 1:u32 then high:bool(R_EBX:u32) ^ R_CF:bool
    else unknown "OF <- undefined":bool
R_SF:bool =
  if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_SF:bool else high:bool(R_EBX:u32)
R_ZF:bool =
  if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_ZF:bool else 0:u32 == R_EBX:u32
R_PF:bool =
  if (R_ECX:u32 & 0x1f:u32) == 0:u32 then R_PF:bool else
5.2 Functions

In this tutorial, we show how to use the get_functions command to extract functions from a binary with debugging symbols. Specifically, we will look at the /bin/ls command. First, make sure you are using a 32-bit executable; BAP does not support x86-64 yet. If we know the name of a function we want to extract, such as hash_table_ok, we can extract it using get_functions -unroll /bin/ls bap hash_table_ok. This will
produce a file called baphash_table_ok.il. We used the \texttt{-unroll} option to remove loops by unrolling; this is important when generating VCs, because generating VCs is generally only possible for acyclic programs. If we wanted to see if the function can return zero, we could use \texttt{topredicate}:

\texttt{topredicate -q -il baphash_table_ok.il -post \textquote{R_EAX:u32 == 0:u32} -stp-out /tmp/f -solve.}

\section{Concrete Evaluation}

In this tutorial, we will explain how to raise a binary to the BAP IL, and then evaluate the resulting code inside of BAP.

Let's start with the following simple C program:

\begin{verbatim}
int g(int y) {
    if (y == 42) { return 42; } else { return -1; }
}

int main(char **argv, int argc) {
    return g(42);
}
\end{verbatim}

We can compile this source file (named test.c) to a binary using \texttt{gcc -static test.c -o test} (note if you are on an x64 machine you may need to add the \texttt{-m32} option). We use the \texttt{-static} option of gcc here because the BAP evaluator needs to know what code is located at each memory location. This means that BAP does not currently support self-modifying code, or dynamic linking (unless the user explicitly tells BAP the mapping of memory addresses to binary code).

Once we have a binary, we can then raise it to the BAP IL by executing \texttt{toil -bin test -o test.il}. We can now disassemble the binary to see where main begins and ends – we will need this to tell BAP where to start and stop executing. Here is a possible disassembly of the main function.

\begin{verbatim}
08048267 <main>:
  8048267: 55 push %ebp
  8048268: 89 e5 mov %esp,%ebp
  804826a: 83 ec 04 sub $0x4,%esp
  804826d: c7 04 24 2a 00 00 00 movl $0x2a,(%esp)
  8048274: e8 d7 ff ff ff call 8048250 <g>
  8048279: c9 leave
\end{verbatim}

38
For this sample disassembly, we can begin execution at address 0x8048267. Likewise, we can halt execution at 0x804827a. To make BAP halt execution, we’ll manually add a halt command to the lifted IL. For this example, you could search for the string `addr 0x804827a` in test.il, which marks the beginning of the ret instruction in the lifted IL. Inserting `halt R_EAX:u32` after the address and label statements will cause the evaluator to halt and return the value in eax. For instance:

```
addr 0x804827a @asm "ret   
label pc_0x804827a
halt R_EAX:u32
```

Now, issue `ileval -il test.il -eval-at mainaddr`, where mainaddr is the address to start from. BAP will evaluate the IL program. If debugging output is enabled (by setting the `BAP_DEBUG_MODULES` environment variable to `SymbEval`), it will print each statement as it is evaluated. After halting, BAP will print the return value, which is eax. Unsurprisingly, the returned value of eax is 42.

Now let’s begin execution starting from the call instruction in main (0x8048250). This will allow us to modify the input to the function g. The `-init-var` and `-init-mem` options allow the user to specify an initial value for a variable (register) or memory address respectively. To replicate what we just did and use an input of 42, we can use the following the command `ileval -il test.il -init-mem R_ESP:u32 42:u32 -eval-at 0x8048274`. Notice that this begins execution at the call instruction instead of the beginning of main. Unsurprisingly, eax has a final value of 42 again. If we instead change the memory pointed to by esp to 43, by `ileval -il test.il -init-mem R_ESP:u32 43:u32 -eval-at 0x8048274`, eax has a final value of -1.

### 5.4 Traces

In this section, we will explain how to use our PIN-based trace recording tool to record a trace. We then show how to perform common trace analysis.

```
804827a:   c3         ret
804827b:   90         nop
804827c:   90         nop
804827d:   90         nop
804827e:   90         nop
804827f:   90         nop
```

804827a:   c3         ret
804827b:   90         nop
804827c:   90         nop
804827d:   90         nop
804827e:   90         nop
804827f:   90         nop
CHAPTER 5. EXAMPLES

tasks.

5.4.1 Setup

The PIN trace tool is located in the pintraces/ directory. Because of licensing reasons, we cannot distribute PIN with the trace tool. The PIN tool cannot be built until PIN has been extracted inside of the BAP directory. In the following, we assume that $BAPDIR$ is set to the BAP directory. For example, if you extracted BAP to /home/user/Downloads/bap-x.y, then you should replace $BAPDIR$ below with /home/user/Downloads/bap-x.y. Once downloaded, PIN should be extracted to $BAPDIR/pin$. On Linux, running the ./getpin.sh script from the $BAPDIR/pintraces directory will automatically download and extract PIN for Linux; the user is responsible for accepting the PIN license agreements.

On Windows, the process is more complicated. We usually test with Windows 7 and Windows XP SP3, but expect NT, 2000, 2003, Vista, and 7 to work. First, make sure that $BAPDIR$ contains no spaces. Then, install GNU Make for Windows (http://gnuwin32.sourceforge.net/packages/make.htm) and Visual C++ 2010 Express (http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express). Make sure to add the directory containing make (the default is C:\Program Files\GnuWin32\bin) to the Windows Path environment variable. Also download Pin and extract it to the $BAPDIR/pin$ directory. Next, we need to upgrade the Visual Studio project for Google’s protobuf library, which our PIN tool depends on.

To do this, navigate to the $BAPDIR/libtracewrap/libtrace/protobuf/vsprojects directory and open protobuf.sln. When the Visual Studio Conversion Wizard appears, click Finish and close the summary window. It is normal for the conversion process to have warnings, but not errors. Next, right click on the libprotobuf project in the Solution Explorer, and select Properties. Select the Release configuration at the top of the dialog box, and then navigate to the C/C++ Code Generation settings. Change the Runtime Library to Multi-threaded (/MT). Finally, close Visual Studio and save the changes to the project. Open a Visual Studio command prompt, navigate to $BAPDIR/libtracewrap/libtrace/src/cpp and run make -f Makefile.windows to build protobufs and the libtrace library.

The PIN tool itself can be built by executing make in the $BAPDIR/pintraces directory on both Windows and Linux. After compilation, the PIN tool

\[1\] This can be done automatically at the command line using devenv.exe, but this tool is only part of Visual Studio proper; it does not come with the Express versions.

\[2\] If you skip this step, you will get unresolved symbol errors while linking the PIN tool.
5.4. TRACES

should exist in

$BAPDIR/pintraces/obj-ia32/gentrace.so (or gentrace.dll on Windows).

In the rest of the chapter, we will assume Linux is being used; most inter-
action with the trace tool is the same.

5.4.2 Recording a trace

To see the command line options to the trace tool, execute

$BAPDIR/pin/pin -t \
$BAPDIR/pintraces/obj-ia32/gentrace.so -help -- /bin/ls

By default, the trace tool will only log instructions that are tainted, i.e.,
those that depend on user input. The options that begin with taint are used
to mark various data as being user input. For instance, -taint-files readme
marks the file readme as being user input.

We will record a trace of a simple buffer overflow. Run

echo 'helloooooooooooooooooooooo' > readme

to create the input file. Then run

make -C $BAPDIR/pintraces/examples/bof1/

and

$BAPDIR/pin/pin -t \
$BAPDIR/pintraces/obj-ia32/gentrace.so -taint-files readme -taint-files /bin/ls

The PIN tool will output many debugging messages; this is normal. If
the trace tool detected the buffer overflow, it will print “stack smashing
detected” near the end of the logs. At this point, there should be a trace
file ending with suffix bpt in the current working directory. In the following
commands, we assume this file is named trace.bpt.

To lift the trace data and print it, run

iltrans -serializedtrace trace.bpt -pp-ast /dev/stdout

It is also possible to concretize the trace, which removes jumps and
performs memory concretization, by executing
iltrans -serializedtrace trace.bpt -trace-concrete -pp-ast /dev/stdout

Adding the -trace-check option before -trace-concrete causes BAP to compare its internal evaluator’s notion of state with the actual values recorded in the trace. It can be used to check for bugs in the IL. Finally, running

iltrans -serializedtrace trace.bpt -trace-formula f

will symbolically execute the trace and output the generated verification condition to the file f. This can then be solved with stp to find satisfying answers to the trace.
Chapter 6

Related Work

Other Binary Analysis Platforms. BAP is designed to 1) have a formal, well-defined IL, 2) explicitly expose the semantics of complex assembly in terms of the simpler BAP IL, and 3) be easy to re-target. There are several other binary analysis platforms which may have some similarity to BAP, do not fulfill all requirements.

Phoenix is a program analysis environment developed by Microsoft as part of their next generation compiler [20]. One of the Phoenix tools allows Microsoft compiled code to be raised up to a register transfer language (RTL). A RTL is a low-level IR that resembles an architecture-neutral assembly.

Phoenix differs from BAP in several ways. First, Phoenix can only lift code produced by a Microsoft compiler. Second, Phoenix requires debugging information, thus is not a true binary-only analysis platform. Third, Phoenix lifts assembly to a low-level IR that does not expose the semantics of complicated instructions, e.g., register status flags, as part of the IR [21]. Fourth, the semantics of the lifted IR, as well as the lifting semantics and goals, are not well specified [21], and thus not suitable for our research purposes.

The CodeSurfer/x86 platform [7] is a proprietary platform for analyzing x86 programs. At the core of CodeSurfer/x86 is a value-set analysis (VSA) [6]. We have also implemented VSA in BAP. CodeSurfer/x86 was not made available for comparison, thus we do not know whether it supports a well-defined IL, is re-targetable, or what other comparable aspects it may share with BAP.
**Decompilation.** Decompilation is the process of inverting the compilation of a program back to the original source language. Generally, the goal of decompilation is to recover a valid program in the original source language that is semantically equivalent (though need not be exactly the same) as the original source program.

Program analysis is often an important component in the decompilation process. Cifuentes has proposed using data and control flow analyses as part of decompilation [11]. Van Emmerik has shown that analysis on SSA flow graphs is helpful in a variety of tasks such as reconstructing types and resolving indirect jumps [14]. BAP also has SSA; thus it should be straightforward to implement Van Emmerik’s algorithms. Mycroft has proposed type inference for recovering C types during decompilation [23]. Adding type inference is a possible future direction for BAP.

**Binary Instrumentation.** Binary instrumentation is a technique to insert extra code into a binary that monitors the instrumented program’s behavior (e.g., [5,17,19,24,26,27,30]). Instrumentation is performed by inserting jumps in the original binary code to the instrumentation code. The instrumentation code then jumps back to the original code after executing. The instrumentation code must make sure that the execution state is the same before and after the jump in order for the instrumentation to be transparent.

Although many binary instrumentation tools provide a limited amount of program analysis, the end-goal is instrumentation, not facilitating analyses. For example, Pin [19] calculates register liveness information. However, general static analysis is outside the scope of such tools. For instance, instrumentation tools generally do not expose the semantics of all instructions such as register status flag updates.

**Other Program Analysis Platforms.** BAP shares many of the same goals, such as modularity and ease of writing correct analyses, as other program analysis platforms. For example, CIL [25] and SUIF [3] are both excellent platforms for analyzing C code. However, using higher-level program analysis platforms is inappropriate for binary code because binary code is fundamentally different. While higher-level languages have types, functions, pointers, loops, and local variables, assembly has no types, no functions, one globally addressed memory region, and goto’s. The BAP IL and surrounding platform is designed specifically to meet the challenges of faithfully analyzing assembly.
Chapter 7

Credits

See http://bap.ece.cmu.edu for a list of current and former BAP developers.
Bibliography


