BAP: A Binary Analysis Platform

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

Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA, USA

Abstract. BAP is a publicly available infrastructure for performing program verification and analysis tasks on binary (i.e., executable) code. In this paper, we describe BAP as well as lessons learned from previous incarnations of binary analysis platforms. BAP explicitly represents all side effects of instructions in an intermediate language (IL), making syntax-directed analysis possible. We have used BAP to routinely generate and solve verification conditions that are hundreds of megabytes in size and encompass 100,000's of assembly instructions.

1 Introduction

Program analysis of binary (i.e., executable) code has become an important and recurring goal in software analysis research and practice. Binary code analysis is attractive because it offers high fidelity reasoning of the code that will actually execute, and because not requiring source code makes such techniques more widely applicable.

BAP, the Binary Analysis Platform, is the third incarnation of our infrastructure for performing analysis on binary code. Like other platforms such as CodeSurfer/x86 [3], McVeto [15], Phoenix [11], and Jakstab [9], BAP first disassembles binary code into assembly instructions, lifts the instructions to an intermediate language (IL), and then performs analysis at the IL level. BAP provides the following salient features:

- BAP makes all side effects of assembly instructions explicit in the IL. This enables all subsequent analyses to be written in a syntax-directed fashion. For example, the core code of our symbolic executor for assembly is only 250 lines long due to the simplicity of the IL. The operational semantics of the IL are formally defined and available in the BAP manual [4].
- Common code representations such as CFGs, static single assignment/three-address code form, program dependence graphs, a dataflow framework with constant folding, dead code elimination, value set analysis [3], and strongly connected component (SCC) based value numbering.
- Verification capabilities via Dijkstra and Flanagan-Saxe style weakest preconditions and interfaces with several SMT solvers. The verification can be performed on dynamically executed traces (e.g., via an interface with Intel's Pin Framework), as well as on static code sequences.
- BAP is publicly available with source code at `http://bap.ece.cmu.edu/`. BAP currently supports x86 and ARM.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 463–469, 2011.

We have leveraged BAP and its predecessors in dozens of security research applications ranging from automatically generating exploits for buffer overflows to inferring types on assembly. A recurring task in our research is to generate logical verification conditions (VCs) from code, usually so that satisfying answers are inputs that drive execution down particular code paths. Generating VCs that are actually solvable in practice is important; we routinely solve VCs hundreds of megabytes in size that capture the semantics of 100,000s of assembly instructions using BAP.

In the rest of this paper we discuss these features, how they evolved, compare them to other platforms where possible, and provide examples of how we have used them in various projects.

2 BAP Goals and Related Work

Fully representing the semantics of assembly is more challenging than it would seem. In order to appreciate the difficulty, consider the three line assembly program below. Suppose we want to create a verification condition (VC) that is satisfied only by inputs that take the conditional jump (e.g., to find inputs that take the jump). The challenge is that arithmetic operations set up to 6 status flags, and control flow in assembly depends upon the values of those flags. Simply lifting line 1 to something like `ebx = eax + ebx` does not expose those side effects.

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

The first generation of our binary analysis tools, asm2c, attempted to directly decompile x86 assembly to C, and then perform all software analysis on the resulting C code. asm2c left instruction side effects implicit, which made it difficult to analyze control flow. Other binary tools such as instrumentors, disassemblers, and editors (e.g., DynInst [13], Valgrind [12], and Microsoft Phoenix [11]) also did not represent these side effects explicitly.

Our next incarnation, Vine, was designed to address the problem by explicitly encoding side-effects in the IL. The result is that subsequent analyses and verification could rely upon the IL syntax alone. Vine is significantly more successful than asm2c, and has been used in dozens of research projects (see [5]).[1] Vine used VEX [12] to provide a rough IL for each instruction, which was then augmented by Vine to expose all otherwise-implicit side effects. An important implementation decision was to implement the Vine back-end in OCaml (asm2c was in C++). We found OCaml's language features to be a much better match for program analysis and verification. However, the Vine IL grew over time, lacked a formal semantics for the IL itself, and did not handle bi-endian architectures such as ARM correctly.

BAP is a complete re-design of Vine that encompasses lessons learned from our previous work on binary analysis. The main goals of BAP are: 1) explicitly

[1] Vine is still actively developed at Berkeley under the BitBlaze project [5].

represent all assembly side-effects to allow for syntax-directed analysis; 2) use a simple IL with formally defined semantics; 3) include useful analyses and verification techniques appropriate for binary code (either by design or by adaptation); and 4) allow user-defined analyses. The semantics of the BAP IL is formally defined, which weeded out several bugs from Vine and allowed us to better argue about the correctness of implemented analyses and algorithms. The IL also adds primitives to handle instruction issues discovered in Vine such as bi-endian memory operations, and is simpler overall. In addition to modeling the semantics of instructions explicitly, BAP also exposes the low-level semantics of memory where loads and stores are byte-addressable and thus can result in “overlapping operations”.

An example of the IL produced for Example 1 is (after deadcode elimination):

```
addr 0x0 @asm ”add %eax,%ebx”
t:u32 = R_EBX:u32
R_EBX:u32 = R_EBX:u32 + R_EAX:u32
R_CF:bool = R_EBX:u32 < t:u32
addr 0x2 @asm ”shl %cl,%ebx”
t1:u32 = R_EBX:u32 >> 0x20:u32 - (R_ECX:u32 & 0x1f:u32)
R_CF:bool =
    ((R_ECX:u32 & 0x1f:u32) = 0:u32) & R_CF:bool |
   ~((R_ECX:u32 & 0x1f:u32) = 0:u32) & low:bool(t1:u32)
addr 0x4 @asm ”jc 0x000000000000000a”
cjmp R_CF:bool, 0xa:u32, ”nocjmp0” # branch to 0xa if R_CF = true
label nocjmp0
```

3 BAP Architectural Overview

BAP is divided into front-end and back-end components that are connected by the BAP intermediate language (IL), as shown in Figure 1. The front end is responsible for lifting binary code for the supported architectures to the IL. The back-end implements our program analyses and verifications for low-level code.

The front end reads binary code from an execution trace or a region of a binary executable. When lifting instructions from a binary, BAP uses a linear sweep disassembly algorithm. The user or an analysis is responsible for directing BAP to properly aligned instructions. The result of lifting is an IL program.

An abbreviated definition of the IL syntax is shown in Table 1; the full IL syntax and semantics are provided at [4]. The `special` statement indicates a system call or other unmodeled behavior. Other statements have their obvious meaning. All expressions in BAP are side-effect free. The `unknown` expression indicates an unknown value; for instance, we use this to model the contents of

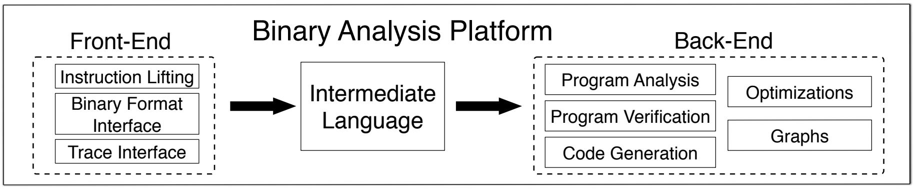

Fig. 1. The BAP binary analysis architecture and components

Table 1. An abbreviated syntax of the BAP IL

program	::=	*stmt**
stmt	::=	*var* := *exp* | `jmp` *exp* | `cjmp` *exp*,*exp*,*exp* | `assert` *exp*
		| `label` *label_kind* | `addr` *address* | `special` *string*
exp	::=	`load`(*exp*, *exp*, *exp*, τ_{reg}) | `store`(*exp*, *exp*, *exp*, *exp*, τ_{reg}) | *exp* $\Diamond_b$ *exp*
		| $\Diamond_u$ *exp* | *var* | `lab`(*string*) | *integer* | `cast`(*cast_kind*, τ_{reg}, *exp*)
		| `let` *var* `=` *exp* `in` *exp* | `unknown`(*string*, τ)

registers having an undefined state after a specific instruction (e.g. the `AF` flag after `shl`). The semantics of `load`$(e_1, e_2, e_3, \tau_{\text{reg}})$ is to load from the memory specified by e_1 at address e_2. e_3 tells us the endianness to use when loading bytes from memory, which can vary at runtime on ARM. τ_{reg} tells us how many bytes to load. `store` is similar, but takes an additional parameter to specify the stored value.

The BAP IL can be transformed into other useful representations. One example is static single assignment (SSA) [1] form. SSA form makes use-def and def-use chains explicit in syntax, and enforces the use of three-address code. These changes often make it significantly easier to implement new analyses and optimizations.

Once a binary is lifted to the BAP IL, it can be analyzed by the BAP back-end. The BAP back-end consists of program analyses and transformations. We discuss these in more detail in Section 4.

Usage. Users are expected to use BAP's front-end to lift binary code to IL form, and then to interact with the analyses and transformations in the back-end. Users can use BAP command line utilities out of the box to perform standard operations. For instance, users can use the `iltrans` tool to create a pipeline of actions that 1) converts an IL program to SSA form; 2) applies all BAP optimizations; 3) converts back to IL form; 4) outputs a verification condition (VC) for the optimized program.

BAP can also be extended programatically. New analyses can build on existing analyses and transformations, allowing for modularity and reuse of implemented analyses similar to a source-level compiler architecture.

4 BAP Capabilities

Analyses and Optimizations. Analyses can either be accessed programatically, or via the command line `iltrans` utility. Built in analyses include the ability to:

– Compute slices for a source or a chop for a source/sink pair so that subsequent analysis only considers relevant parts of a program. For example, if we are interested in whether integer overflow occurs for a particular variable we can reason about the slice of statements affecting (backwards slicing) or affected by (forward slicing) that variable.

- Optimize the IL. Optimizations are an important part of the BAP framework for several reasons. First, the IL makes all side-effects explicit by default, many of which may not matter for a particular analysis. Deadcode elimination will remove these. For instance, deadcode elimination will remove OF, SF, ZF, AF, and PF in Example 2 because they are not relevant.

  In our coreutils experiments [8], we found that the use of optimizations resulted in an overall speedup of 4.5x in the time it took to generate and solve formulas, and enabled us to solve 81% of the VCs that could not be solved without optimizations.
- Evaluate the IL. Our evaluator allows us to run a BAP program and examine any dynamic properties. For instance, the evaluator can be used to record control flow, perform randomized testing of a software property, or verify that the IL semantics are consistent with the real program's.

Verification Conditions. BAP can create verification conditions using several methods. A verification condition (VC) is a syntactically generated boolean predicate over a program's input variables that is true if and only if some program property holds over the program's execution on that input. Naturally, a VC is valid if and only if the respective program property holds for all inputs. BAP generates VCs with respect to a postcondition, such that if the formula is true then the program terminates and the postcondition holds.

Built-in methods for generating VCs include:

- Dijkstra's weakest preconditions (WP). The process involves converting the BAP IL, which represents unstructured code, to Dijkstra's guarded command language. The resulting VC is $O(2^n)$ in size where n is the number of IL statements. Other methods produce smaller VCs.
- Efficient weakest preconditions. We implement two algorithms. First, we have implemented Flanagan and Saxe's algorithm, which guarantees the generated VC will be only $O(n^2)$ in size where n is the number of IL statements. Second, we have developed and proved correctness of a variant of Flanagan and Saxe that can be run in the forward direction [8].
- Forward symbolic execution [14]. Symbolic execution is built into BAP's evaluator.
- Direct (API) and filesystem bindings to STP [7], as well as the ability to interact via the filesystem with SMTLIB1 compliant decision procedures.

5 Applications

We have used the BAP toolchain for a number of binary analysis and verification tasks. Due to space, please refer to [6] for a full list. Example applications are:

- We designed and performed type reconstruction on compiled C programs in a system called TIE [10]. TIE analyzes each memory access in x86 to find variable locations (similar to VSA [2]), creates a system of type constraints based upon variable usage, and solves for a typing on all variables.

- We evaluated the performance of VC generation algorithms by checking VCs for leaf functions in GNU coreutils [8]. For instance, we tested each function to see if the overflow flag could be set, or if the return address could be overwritten[2]. For each condition, we generated a VC and checked its validity with standard SMT solvers (CVC3 and Yices).
- Perform binary-only symbolic execution. We are able to lift TEMU [5] instruction traces to our IL, add constraints on the input, e.g., to find inputs where a safety property breaks, generate an input that takes a specific branch in the trace, and so on. We have used this to perform automatic patch-based exploit generation, malware analysis, and other security-related tasks [5, 6].

6 Limitations

BAP currently supports subsets of the x86 and ARM ISAs. Some features, like floating point and privileged instructions are unsupported. It is not possible to prove the correctness of BAP's lifting code correct because the semantics of the x86 ISA is not formally defined. Instead, we use random testing to identify any differences between the semantics of our lifted IL and behavior on a real processor.

BAP's lifting process expects to be pointed to an aligned sequence of instructions. Thus, the user must identify code locations. This can be done manually, by relying on symbol data, or by using a recursive descent analysis (such as IDA Pro). Lifting also assumes that code is static. BAP's execution trace feature can be used to reason about dynamic code.

Some analyses require indirect jumps to be resolved to concrete locations. For instance, it is not possible to generate VCs using weakest preconditions in the presence of unresolved indirect jumps, since weakest precondition is a static analysis. (It is still possible to use dynamic symbolic execution, however.)

7 Conclusion

BAP is a flexible binary analysis framework that enables program analysis and verification on binary code. BAP explicitly represents side effects of instructions in a simple, formally defined IL. A number of analyses, optimizations, and verification techniques are already built into BAP, and adding new ones is easy. The source code for BAP is periodically released at `http://bap.ece.cmu.edu`.

References

1. Appel, A.: Modern Compiler Implementation in ML. Cambridge University Press, Cambridge (1998)
2. Balakrishnan, G.: WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Science Department, University of Wisconsin at Madison (August 2007)

[2] We chose these postconditions because they are non-trivial and can be applied to all functions.

3. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: Codesurfer/x86 - a platform for analyzing x86 executables. In: Proceedings of the International Conference on Compiler Construction (April 2005)
4. Binary Analysis Platform (BAP), `http://bap.ece.cmu.edu`
5. BitBlaze binary analysis project (2007), `http://bitblaze.cs.berkeley.edu`
6. Brumley, D.: `http://security.ece.cmu.edu`
7. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proceedings of the Conference on Computer Aided Verification, pp. 524–536 (July 2007)
8. Jager, I., Brumley, D.: Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab (February 2010)
9. Kinder, J., Veith, H.: Jakstab: A static analysis platform for binaries. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 423–427. Springer, Heidelberg (2008)
10. Lee, J., Avgerinos, T., Brumley, D.: TIE: Principled reverse engineering of types in binary programs. In: Proceedings of the Network and Distributed System Security Symposium (February 2011)
11. Microsoft. Phoenix framework, `http://research.microsoft.com/phoenix/` (url checked April 21, 2011)
12. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. In: Proceedings of the Third Workshop on Runtime Verification, Boulder, Colorado, USA (July 2003)
13. Paradyn/Dyninst. Dyninst: An application program interface for runtime code generation, `http://www.dyninst.org` (url checked April 21, 2011)
14. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 317–331 (May 2010)
15. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 288–305. Springer, Heidelberg (2010)