5 Static Analysis of Binaries

Static binary analysis is more difficult than if source code is available. In many cases, the analyses are unsound and behaviours are omitted to make problems feasible. Heuristics may be required to separate code and data in a disassembly or pointer behaviour may be weakly modelled to make statically analysing programs feasible. Nevertheless, static analysis of binaries is an important area of research with a number of practical applications including the detection of software theft and the classification and detection of malware. This chapter examines static analysis of binaries with the intent that properties and features of binary programs can be extracted to create useful birthmarks for software similarity and classification.

Keywords: Disassembly, intermediate language, control flow reconstruction, decompilation.

5.1 Disassembly

Disassembly is the process of translating machine code to assembly language [1]. This is typically the first stage of a static analysis. Static disassembly parses the entire binary image statically without execution. In static disassembly, there are two main algorithms. In the Linear Sweep algorithm, the instructions are disassembled one instruction after another, starting from the beginning of code. The disadvantage of this method is that data introduced into instruction stream may be erroneously disassembled.

The other main algorithm to perform disassembly is the Recursive Traversal algorithm. This algorithm decodes each instruction following the order of the control flow. This resolves the issue of embedded data, but may miss decoding instructions that are the target of indirect jumps or other situations when it is hard to resolve control flow statically.

Speculative Disassembly attempts to remedy the problems of the Recursive Traversal algorithm problem by first performing the Recursive Traversal, and then performing a Linear Sweep in regions that are not decoded.

Disassembly results in the following data.

$$disassembly = \{address, opcode, operand_1, ..., operand_n\}$$

```
disassemble_program(program)
{
  address = disassemble_linear_sweep(
    start(program), end(program))
}

disassemble_linear_sweep(start, end) {
  address = start
  while (address < end) {
    instruction = Disassemble(program, address)
    if (error) {
      address += 1;
    } else {
      disassembly[address] = instruction;
      address += length(instruction);
    }
  }
}
```

Fig. 5.1 Linear sweep disassembly.

5.2 Intermediate Code Generation

A simple approach to transforming assembly into an intermediate language is to translate each instruction without maintaining intermediate state. This approach has been used successfully in the Reverse Engineering Intermediate Language (REIL) [2]. Other popular intermediate languages are Vex as used in the Valgrind binary instrumentation framework [3] and Vine as used in the BitBlaze project [4]. An example to translate native assembly into three address code is shown below.

$$native\_assembly\_instruction \rightarrow (TAC_1,..TAC_n)$$

5.3 Procedure Identification

An important stage in reconstruction the control flow of an executable is identifying procedures. There are roughly four approaches that can be employed.

```
disassemble_program(program) {
  disassemble(entry_point(program))
}
disassemble_recursive_traversal(address) {
  while (has_address(program, address)) {
    if (disassembly[address] not null)
     return
    instruction = Disassemble(program, address)
    if (error)
      return
    disassembly[address] = instruction
    if (is_return_instruction(instruction))
      return
    if (is_transfer_instruction(instruction))
       disassemble(transfer_target(instruction);
    address += length(instruction);
  }
}
```

Fig. 5.2 Recursive traversal disassembly.

```
disassemble_speculative(program) {
  disassemble_recursive_traversal(entry_point(program))
  for all intervals in
    [start(program), end(program)] and not in disassembly
  {
    disassemble_linear_sweep(
      start(interval), end(interval))
  }
}
```

Fig. 5.3 Speculative disassembly.

- Using object file format information (e.g., symbols and exports)
- Using static targets of call site $F = \{f \mid (address, call\_direct, f) \in disassembly\}$
- Using idioms to identify procedure prologues

```
disassemble_procedure(address) {
  while (has_address(program, address)) {
    if (disassembly[address] not null)
     return
    instruction = Disassemble(program, address)
    if (error)
      return
    disassembly[address] = instruction
    if (is_return_instruction(instruction))
      return
    if (is_transfer_instruction(instruction)
        and not is_call_instruction(instruction))
      disassemble_procedure(transfer_target(instruction);
    address += length(instruction);
  }
}
```

Fig. 5.4 Procedure disassembly.

- Using static analysis and data flow analysis to reconstruct indirect call targets

The main hindrance to generating accurate representations is when a program uses indirect branches and procedure calls. The analysis of indirect targets requires data flow analysis. A number of approaches have been employed [5-7]. Using idioms to identify procedures requires string matching algorithms to identify common byte sequences.

5.4 Procedure Disassembly

Procedures consist of a body of instructions which must be recovered from the disassembly. The algorithm is a very slight variation of the recursive traversal disassembly algorithm. The difference is that inter procedural control flow is not traversed.

5.5 Control Flow Analysis, Deobfuscation and Reconstruction

Control flow analysis is more difficult on binaries because of the difficultly in separating code and data. Likewise, the presence of indirect branch and call targets in assembly language makes precisely determining the static control flow undecidable.

The simplest approach is to ignore indirect targets completely. The edges of the graphs representing the call graph control flow can be constructed by connecting the call site to the static call target. For control flow graphs the approach is similarly applied to branch targets.

Control flow may also be obfuscated. An opaque predicate [8] is a predicate that always evaluates to the same result. An opaque predicate is constructed so that it is difficult for an analyst or automated analysis to know the predicate result. Opaque predicates can be used to insert superfluous branching in a binary's control flow. They can also be used to assign variables values which are hard to determine statically. The use of opaque predicates is primarily for code obfuscation, and to prevent understanding by an analyst or automated static analysis.

The presence of opaque predicates in a control flow graph reduces the accuracy of the graph because of misleading branch targets. In [9] it was proposed to use the program analysis technique of abstract interpretation to detect specific classes of opaque predicate algorithms.

5.6 Pointer Analysis

Pointer and alias analysis tries to determine the variables that a pointer may point to. In assembly this problem is difficult. A conservative approach to alias analysis of assembly using datalog constraints was proposed in [10], however this work was to introduce formal rigour and is not practical to deploy. Value-Set Analysis [11] has been proposed as an alias analysis, suitable for binary programs and assembly language. Value-Set Analysis has been used in malware detection [12] and the automated static unpacking of malware [13].

5.7 Decompilation of Binaries

Decompilation [14] is the process of recovering source code from executable binaries. In general, decompilation can be seen as a form of static analysis of a binary that recovers additional information from its intermediate representation. Research connecting the type of static analysis a compiler performs to the requirements of a decompiler was proposed in [14] and [15].

5.7.1 Condition Code Elimination

In Instruction Set Architectures such as x86, many arithmetical instructions modify a status flag or condition code. For example, determining if two variables are equal is divided into two computations. An arithmetic instruction over the two variables that sets a condition code, and then a branch based on the resulting condition code. Decompilation requires these two computations be reduced to one conditional test.

An approach to solve this is by maintaining a reaching definition of the various conditions code set by each arithmetic instruction. At the point of a conditional branch based on the condition code, the reaching definitions are combined into a single condition.

5.7.2 Stack Variable Reconstruction

Stack variable reconstruction transforms variables allocated on the stack into native variables in the intermediate representation. The stack can be accessed in two main ways. The first method is by referencing variables relative to the top of the stack, or stack pointer. The second method accesses the stack relative to the frame pointer. The frame pointer is unique for each procedure or activation record. It points to the top of the stack as set on function entry. During procedure execution the stack pointer may change, but the frame pointer remains constant. This simplifies access to variables on the stack and is often used in debug builds of application. It is clear that for a decompiler to be effective, it must handle both methods of accessing the stack. Both frame and stack based addressing may be intermixed in real life applications.

Another complication to using the stack pointer is that callees may or may not change the stack pointer. It is the responsibility of the caller to push arguments onto the stack, but the callee may or may not unwind these arguments based on the calling convention being used.

One approach [16] to reconstruct stack based variables takes advantage of the fact that in compiled programs, the position of the stack pointer in each basic block remains constant. The stack pointer can be modified within a basic block when calls are made or values or pushed and popped on or from the stack. Using this information, a set of constraints over the control flow graph can describe the stack pointer. Solving the constraints identifies the relative position of the stack pointer at the entry and exit of each basic block. Frame pointer relative addressing uses fixed offsets from the top of the stack at the beginning of the procedure, and knowing the position of the stack pointer at each basic block enables knowing exactly which memory location on the stack is being referenced. This enables a unified approach to modelling stack and frame based addressing.

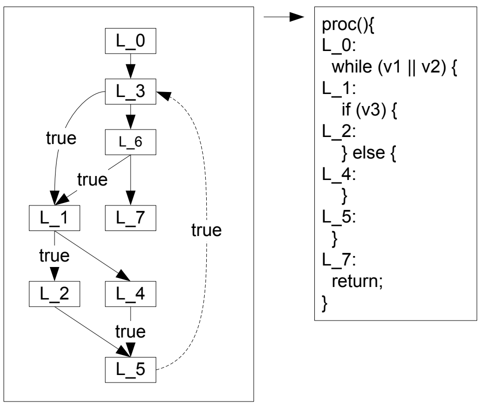
Pointers and arrays complicate the process of stack variable reconstruction. In these cases, the stack variable may only be referencing the beginning of an array or pointing to the beginning of the object. Heuristics must be used to estimate the size of the object. An approach to estimate this is by looking at the size of the stack frame or looking at the next adjacent stack reference to predict a bounds on the object in question.

5.7.3 Preserved Register Detection

A typical problem that arises is determining if the register is modified in the life time of a procedure. If the register is used in procedure, but maintains its original value once returning from the procedure's callsite then the register is preserved. The process of preserving a register is to copy the register into a temporary variable and then restore it before leaving the function. Detecting preserved registers is important in the process of identifying which registers are arguments or return values from a procedure.

Data flow analysis and a suitable intermediate representation can help solve the preserved register problem. If we ignore calls within a procedure, we can identify a preserved register by the fact that the reaching definitions for that register at each function exit, is the value of a copy of the register on function entry. To determine where the value is copied on entry to the function we can use a liveness analysis to identify where the register is used and check that instruction for a copy instruction.

This process of identifying preserved registers requires that local variable reconstruction be performed. The reason is that the temporary variable used to save a copy of the preserved register is typically represented by a local variable.



```
proc(){
L_0:
  while (v1 || v2) {
L_1:
    if (v3) {
L_2:
    } else {
L_4:
    }
L_5:
  }
L_7:
  return;
}
```

Fig. 5.5 A control flow graph and its linearized form.

5.7.4 Procedure Parameter Reconstruction

The parameters to procedures may be passed on the stack, or passed via registers. The return values are typically passed by registers. The exact semantics are defined the calling convention on a particular procedure. The arguments used by a procedure can be determined by the procedure accessing variables outside the current stack frame. Once the arguments are known, at call sites, the stack is statically unwound to the required depth to retrieve them.

Registers may also be passed as arguments. Ignoring calls, arguments are registers that are live on procedure entry that aren't preserved. To take into account calls, the analysis is performed on inner calls first as defined by their depth first order in the call graph. Recursive calls require further analysis.

5.7.5 Reconstruction of Structured Control Flow

A standard technique in decompilation is transforming a control flow graph into higher level structured control flow [17,18,14]. This is the process of structuring. Identifying conditions, loops, and parts of the control flow graph that cannot be structured is required. Conditions may be compound conditional statements involving conjunction and disjunction. The higher the quality of structuring means the less the number of gotos in the generated code. Some graphs cannot be structured and the reducibility of the graph identifies these cases.

Structuring of control flow graphs was proposed in [19,20] to generate string signatures that were later used to identify malware variants.

5.7.6 Type Reconstruction

Type information is lacking from binaries. Reconstruction of types enables higher quality code in the decompiled output. An approach to type reconstruction using the unification algorithm was proposed in [21]. A data flow analysis approach based on lattices and using single static analysis was proposed in [15].

5.8 Obfuscation and Limits to Static Analysis

It is known that perfectly precise disassembly is undecidable [22]. Branch targets can be indirect, and precise understanding of those run-time values can be problematic. In [23] an analysis of some limits to static analysis of malware were identified. The use of opaque predicates was shown to confound the problem of precise program representation. Determining whether two programs are semantically equivalent is also known to an undecidable problem which is why for example malware detection is often based on heuristic and unsound solutions. Likewise, perfect decompilation, for all possible binaries, is undecidable. If the binary does not originate from high level source then it is unlikely decompilation will give meaningful results.

5.9 Research Opportunities

Decompilation presents potential research opportunities when combined with other techniques such as static analysis or malware classification. Very little research has been performed on decompilation-based applications. The main application of decompilation thus far has been source code recovery. However, the high level information it recovers makes it a suitable abstraction for useful software features.

References

1. Kruegel C, Robertson W, Valeur F, Vigna G Static disassembly of obfuscated binaries. In: USENIX Security Symposium, 2004. pp 18-18
2. Dullien T, Porst S (2009) REIL: A platform-independent intermediate representation of disassembled code for static code analysis. CanSecWest,
3. Nethercote N, Seward J (2003) Valgrind A Program Supervision Framework. Electronic Notes in Theoretical Computer Science 89 (2):44-66
4. Song D, Brumley D, Yin H, Caballero J, Jager I, Kang M, Liang Z, Newsome J, Poosankam P, Saxena P (2008) BitBlaze: A new approach to computer security via binary analysis. Information Systems Security:1-25
5. Daniel K, stner, Stephan W (2002) Generic control flow reconstruction from assembly code. SIGPLAN Not 37 (7):46-55. doi:http://doi.acm.org/10.1145/566225.513839
6. Theiling H (2000) Extracting safe and precise control flow from binaries. Paper presented at the Proceedings of the Seventh International Conference on Real-Time Systems and Applications,
7. Johannes K, Florian Z, Helmut V (2009) An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. Paper presented at the Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation, Savannah, GA,
8. Cullen L, Saumya D (2003) Obfuscation of executable code to improve resistance to static disassembly. Paper presented at the Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C., USA,
9. Dalla Preda M, Madou M, De Bosschere K, Giacobazzi R Opaque predicates detection by abstract interpretation. Algebraic Methodology and Software Technology:81–95
10. Brumley D, Newsome J (2006) Alias analysis for assembly. Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006,
11. Balakrishnan G, Reps T, Melski D, Teitelbaum T (2007) Wysinwyx: What you see is not what you execute. Verified Software: Theories, Tools, Experiments:202-213
12. Leder F, Steinbock B, Martini P Classification and Detection of Metamorphic Malware using Value Set Analysis. In: Proc. of 4th International Conference on Malicious and Unwanted Software (Malware 2009), Montreal, Canada, 2009.
13. Debray KCS, Townsend TKG (2009) Automatic Static Unpacking of Malware Binaries. Paper presented at the Working Conference on Reverse Engineering - WCRE,
14. Cifuentes C (1994) Reverse compilation techniques. Queensland University of Technology,
15. Van Emmerik MJ (2007) Static single assignment for decompilation. The University of Queensland,
16. Hex-Rays S (2008) IDA Pro Disassembler.
17. Moretti E, Chanteperdrix G, Osorio A (2001) New algorithms for control-flow graph structuring. Paper presented at the Software Maintenance and Reengineering,
18. Wei T, Mao J, Zou W, Chen Y (2007) Structuring 2-way branches in binary executables. Paper presented at the International Computer Software and Applications Conference,
19. Cesare S, Xiang Y Classification of Malware Using Structured Control Flow. In: 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), 2010.
20. Cesare S, Xiang Y (2010) A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost. In: IEEE 24th International Conference on Advanced Information Networking and Application (AINA 2010), 2010. IEEE,
21. Mycroft A (1999) Type-based decompilation. Lecture notes in computer science:208-223
22. Horspool RN, Marovac N (1979) An approach to the problem of detranslation of computer programs. The Computer Journal 23 (3):223-229
23. Moser A, Kruegel C, Kirda E Limits of static analysis for malware detection. In: Annual Computer Security Applications Conference (ACSAC), 2007.