1

Static Disassembly and Code Analysis

Giovanni Vigna

Reliable Software Group
University of California, Santa Barbara

Summary. The classification of an unknown binary program as malicious or benign requires two steps. In the first step, the stream of bytes that constitutes the program has to be transformed (or disassembled) into the corresponding sequence of machine instructions. In the second step, based on this machine code representation, static or dynamic code analysis techniques can be applied to determine the properties and function of the program.

Both the disassembly and code analysis steps can be foiled by techniques that obfuscate the binary representation of a program. Thus, robust techniques are required that deliver reliable results under such adverse circumstances. In this chapter, we introduce a disassemble technique that can deal with obfuscated binaries. Also, we introduce a static code analysis approach that can identify high-level semantic properties of code that are difficult to conceal.

1.1 Introduction

Code analysis takes as input a program and attempts to determine certain characteristics of this program. In particular, the goal of security analysis is to identify either malicious behavior or the presence of security flaws, which might be exploited to compromise the security of a system. In this chapter, we focus particularly on the security analysis of binary programs that use the Intel x86 instruction set. However, many of the concepts can also be applied to analyze code that exists in a different representation.

In the first step of the analysis, the code has to be disassembled. That is, we want to recover a symbolic representation of a program's machine code instructions from its binary representation. While disassembly is straightforward for regular binaries, the situation is different for malicious code. In particular, a number of techniques have been proposed that are effective in preventing a substantial fraction of a binary program from being disassembled correctly. This could allow an attacker to hide malicious code from the subsequent static program analysis. In Section 1.2, we present binary analysis techniques that substantially improve the success of the disassembly process when confronted with obfuscated binaries. Using control flow graph information and statistical methods, a large fraction of the program's instructions can be correctly identified.

Based on the program's machine code, the next step is to identify code sequences that are known to be malicious (or code sequences that violate a given specification of permitted behavior). Often, malicious code is defined at a very low level of abstraction. That is, a specification, or signature, of malicious code is expressed in terms of byte sequences or instruction sequences. While it is efficient and easy to search a program for the occurrence of specific byte strings, such *syntax-based* signatures can be trivially evaded. Therefore, specifications at a higher level are needed that can characterize the intrinsic properties of a program that are more difficult to disguise. Of course, suitable analysis techniques are required that can identify such higher-level properties. Moreover, these techniques have to be robust against deliberate efforts of an attacker to thwart analysis.

Code analysis techniques can be categorized into two main classes: dynamic techniques and static techniques. Approaches that belong to the first category rely on monitoring execution traces of an application to identify the executed instructions and their actions, or behavior. Approaches that belong to the second category analyze the binary structure statically, parsing the instructions as they are found in the binary image and attempting to determine a (possibly over-approximated) set of all possible behaviors.

Both static and dynamic approaches have advantages and disadvantages. Static analysis takes into account the complete program, while dynamic analysis can only operate on the instructions that were executed in a particular set of runs. Therefore, it is impossible to guarantee that the whole executable with all possible actions was covered when using dynamic analysis. On the other hand, dynamic analysis assures that only actual program behavior is considered. This eliminates possible incorrect results due to overly conservative approximations that are often necessary when performing static analysis.

In Section 1.3, we introduce our static analysis approach to find pieces of code that perform actions (i.e., behave) in a way that we have specified as malicious. More precisely, we describe our application of symbolic execution to the static analysis of binaries.

1.2 Robust Disassembly of Obfuscated Binaries

In this section, we introduce our approach to robust disassembly when facing obfuscated, malicious binaries. The term *obfuscation* refers to techniques that preserve the program's semantics and functionality while, at the same time, making it more difficult for the analyst to extract and comprehend the program's structures. In the context of disassembly, obfuscation refers to transformations of the binary such that the parsing of instructions becomes difficult.

In [13], Linn and Debray introduced novel obfuscation techniques that exploit the fact that the Intel x86 instruction set architecture contains variable length instructions that can start at arbitrary memory address. By inserting padding bytes at locations that cannot be reached during run-time, disassemblers can be confused to misinterpret large parts of the binary. Although their approach is limited to Intel x86 binaries, the obfuscation results against current state-of-the-art disassemblers are remarkable.

In general, disassemblers follow one of two approaches. The first approach, called linear sweep, starts at the first byte of the binary's text segment and proceeds from there, decoding one instruction after another. It is used, for example, by GNU's objdump [8]. The drawback of linear sweep disassemblers is that they are prone to errors that result from data embedded in the instruction stream. The second approach, called recursive traversal, fixes this problem by following the control flow of the program [4, 15]. This allows recursive disassemblers such as IDA Pro [7] to circumvent data that is interleaved with the program instructions. The problem with the second approach is that the control flow cannot always be reconstructed precisely. When the target of a control transfer instruction such as a jump or a call cannot be determined statically (e.g., in case of an indirect jump), the recursive disassembler fails to analyze parts of the program's code. This problem is usually solved with a technique called speculative disassembly [3], which uses a linear sweep algorithm to analyze unreachable code regions.

Linn and Debray's approach [13] to confuse disassemblers are based on two main techniques. First, junk bytes are inserted at locations that are not reachable at run-time. These locations can be found after control transfer instructions such as jumps where control flow does not continue. Inserting junk bytes at unreachable locations should not affect recursive disassemblers, but has a profound impact on linear sweep implementations.

The second technique relies on a branch function to change the way regular procedure calls work. This creates more opportunities to insert junk bytes and misleads both types of disassemblers. A normal call to a subroutine is replaced with a call to the branch function. This branch function uses an indirect jump to transfer control to the original subroutine. In addition, an offset value is added to the return address of the subroutine, which has been saved on the stack as part of the subroutine invocation. Therefore, when the subroutine is done, control is not transfered to the address directly after the call instruction. Instead, an instruction that is a certain number of bytes after the call instruction is executed. Because calls are redirected to the branch function, large parts of the binary become unreachable for the recursive traversal algorithm. As a result, recursive traversal disassemblers perform even worse on obfuscated binaries than linear sweep disassemblers.

When analyzing an obfuscated binary, one cannot assume that the code be generated by a well-behaved compiler. In fact, the obfuscation techniques introduced by Linn and Debray [13] precisely exploit the fact that standard disassemblers assume certain properties of compiler-generated code that can be violated without changing the program's functionality. However, in general, certain properties are easier to change than others and it is not straightforward to transform a binary into a functionally equivalent representation in which all the compiler-related properties of the original code are lost. When disassembling obfuscated binaries, we require that certain assumptions are valid.

First of all, we assume that valid instructions must not overlap. An instruction is denoted as *valid* if it belongs to the program, that is, it is reached (and executed) at run-time as part of some legal program execution trace. Two instructions *overlap* if one or more bytes in the executable are shared by both instructions. In other words, the start of one instruction is located at an address that is already used by another instruction. Overlapping instructions have been suggested to complicate disassembly in [5]. However, suitable candidate instructions for this type of transformation are difficult to find in real executables and the reported obfuscation effects were minimal [13].

The second assumption is that conditional jumps can be either taken or not taken. This means that control flow can continue at the branch target or at the instruction after the conditional branch. In particular, it is not possible to insert junk bytes at the branch target or at the address following the branch instruction. Linn and Debray [13] discuss the possibility to transform unconditional jumps into conditional branches using opaque predicates. Opaque predicates are predicates that always evaluate to either true or false, independent of the input. This would allow the obfuscator to insert junk bytes either at the jump target or in place of the fall-through instruction. However, it is not obvious how to generate opaque predicates that are not easily recognizable for the disassembler. Also, the obfuscator presented in [13] does not implement this transformation.

In addition to the assumptions above, we also assume that the code is not necessarily the output of a well-behaved compiler. That is, we assume that an arbitrary amount of junk bytes can be inserted at unreachable locations. Unreachable locations denote locations that are not reachable at run-time. These locations can be found after instructions that change the normal control flow. For example, most compilers arrange code such that the address following an unconditional jump contains a valid instruction. However, we assume that an arbitrary number of junk bytes can be inserted there. Also, the control flow does not have to continue immediately after a call instruction. Thus, an arbitrary number of padding bytes can be added after each call. This is different from the standard behavior where it is expected that the callee returns to the instruction following a call using the corresponding return instruction. More specifically, in the x86 instruction set, the `call` operation performs a jump to the call target and, in addition, pushes the address following the call instruction on the stack. This address is then used by the corresponding `ret` instruction, which performs a jump to the address currently on top of the stack. However, by redirecting calls to a branch function, it is trivial to change the return address.

Given the assumptions above, we have developed two classes of techniques: general techniques and tool-specific techniques. General techniques are techniques that do not rely upon any knowledge on *how* a particular obfuscator transforms the binary. It is only required that the transformations respect our assumptions. Our general techniques are based on the program's control flow, similar to a recursive traversal disassembler. However, we use a different approach to construct the control flow graph, which is more resilient to obfuscation attempts. Program regions that are not covered by the control flow graph are analyzed using statistical techniques.

An instance of an obfuscator that respects our assumptions is presented by Linn and Debray in [13]. By tailoring the static analysis process against a particular tool, it is often possible to reverse some of the performed transformations and improve the analysis results. For more information on how we can take advantage of tool-specific knowledge when disassembling binaries transformed with Linn and Debray's obfuscator, please refer to [11]. In the following, we only concentrate on the general disassembly techniques.

1.2.1 Function Identification

The first step when disassembling obfuscated programs is to divide the binary into functions that can then be analyzed independently. The main reason for doing so is run-time performance; it is necessary that the disassembler scale well enough such that the analysis of large real-world binaries is possible.

An important part of our analysis is the reconstruction of the program's control flow. When operating on the complete binary, the analysis does not scale well for large programs. Therefore, the binary is broken into smaller regions (i.e., functions) that can be analyzed consecutively. This results in a run-time overhead of the disassembly process that is linear in the number of instructions (roughly, the size of the code segment).

A straightforward approach to obtain a function's start addresses is to extract the targets of call instructions. When a linker generates an ordinary executable, the targets of calls to functions located in the binary's text segment are bound to the actual addresses of these functions. Given the call targets and assuming that most functions are actually referenced from others within the binary, one can obtain a fairly complete set of function start addresses. Unfortunately, this approach has two drawbacks. One problem is that this method requires that the call instructions are already identified. As the objective of our disassembler is precisely to provide that kind of information, the call instructions are not available at this point. Another problem is that an obfuscator can redirect all calls to a single branching function that transfers control to the appropriate targets. This technique changes all call targets to a single address, thus removing information necessary to identify functions.

We use a heuristic to locate function start addresses. More precisely, function start addresses are located by identifying byte sequences that implement typical function prologs. When a function is called, the first few instructions usually set up a new stack frame. This frame is required to make room for local variables and to be able restore the stack to its initial state when the function returns. In the current implementation, we scan the binary for byte sequences that represent instructions that push the frame pointer onto the stack and instructions that increase the size of the stack by decreasing the value of the stack pointer. The technique works very well for regular binaries and also for the obfuscated binaries used in our experiments. The reason is that the used obfuscation tool [13] does not attempt to hide function prologs. It is certainly possible to extend the obfuscator to conceal the function prolog. In this case, our function identification technique might require changes, possibly using tool-specific knowledge.

Note that the partitioning of the binary into functions is mainly done for performance reasons, and it is not crucial for the quality of the results that all functions are correctly identified. When the start point of a function is missed, later analysis simply has to deal with one larger region of code instead of two separate smaller parts. When a sequence of instructions within a function is misinterpreted as a function prolog, two parts of a single function are analyzed individually. This could lead to less accurate results when some intra-procedural jumps are interpreted as inter-procedural, making it harder to reconstruct the intra-procedural control flow graph as discussed in the following section.

1.2.2 Intra-Procedural Control Flow Graph

To find the valid instructions of a function (i.e., the instructions that belong to the program), we attempt to reconstruct the function's intra-procedural control flow graph. A control flow graph (CFG) is defined as a directed graph $G = (V, E)$ in which vertices $u, v \in V$ represent basic blocks and an edge $e \in E : u \rightarrow v$ represents a possible flow of control from u to v. A basic block describes a sequence of instructions without any jumps or jump targets in the middle. More formally, a basic block is defined as a sequence of instructions where the instruction in each position dominates, or always executes before, all those in later positions, and no other instruction executes between two instructions in the sequence. Directed edges between blocks represent jumps in the control flow, which are caused by control transfer instructions (CTIs) such as calls, conditional and unconditional jumps, or return instructions.

The traditional approach to reconstructing the control flow graph of a function works similar to a recursive disassembler. The analysis commences at the function's start address and instructions are disassembled until a control transfer instruction is encountered. The process is then continued, recursively, at all jump targets that are local to the procedure and, in case of a call instruction or a conditional jump, at the address following the instruction. In case of an obfuscated binary, however, the disassembler cannot continue directly after a call instruction. In addition, many local jumps are converted into non-local jumps to addresses outside the function to blur local control flow. In most cases, the traditional approach leads to a control flow graph that covers only a small fraction of the valid instructions of the function under analysis.

We developed an alternative technique to extract a more complete control flow graph. The technique is composed of two phases: in the first phase, an initial control flow graph is determined. In the following phase, conflicts and ambiguities in the initial CFG are resolved. The two phases are presented in detail in the following two sections.

```
    8048000 | 55              push   %ebp                       ¦  function func(int arg) {
    8048001 | 89 e5           mov    %esp, %ebp                 ¦      int local_var, ret_val;
            |                                                   ¦
    8048003 | e8 00 00 74 11  call   19788008 <branch fnct>     ¦      local = other_func(arg);
    8048008 | 0a 05           (junk)                            ¦
            |                                                   ¦
    804800a | 3c 00           cmp    0, %eax                    ¦      if (local_var == 0)
    804800c | 75 06           jne    8048014 <L1>               ¦
    804800e | b0 00           mov    0, %eax                    ¦          ret_val = 0;
    8048010 | eb 07           jmp    8048019 <L2>               ¦      else
    8048012 | 0a 05           (junk)                            ¦
L1: 8048014 | a1 00 00 74 01  mov    (1740000), %eax            ¦          ret_val = global_var;
            |                                                   ¦
L2: 8048019 | 89 ec           mov    %ebp, %esp                 ¦      return ret_val;
    804801b | 5d              pop    %ebp                       ¦
    804801c | c3              ret                               ¦
    804801d | 90              nop                               ¦  }
            |                                                   ¦
         Disassembly of Obfuscated Function                             C Function
```

Fig. 1.1. Example function.

1.2.3 Initial Control Flow Graph

To determine the initial control flow graph for a function, we first decode all possible instructions between the function's start and end addresses. This is done by treating each address in this address range as the beginning of a new instruction. Thus, one potential instruction is decoded and assigned to each address of the function. The reason for considering every address as a possible instruction start stems from the fact that x86 instructions have a variable length from one to fifteen bytes and do not have to be aligned in memory (i.e., an instruction can start at an arbitrary address). Note that most instructions take up multiple bytes and such instructions overlap with other instructions that start at subsequent bytes. Therefore, only a subset of the instructions decoded in this first step can be valid. Figure 1.2 provides a partial listing of all instructions in the address range of the sample function (both in source and assembler format) that is shown in Figure 1.1. For the reader's reference, valid instructions are marked by an `x` in the "*Valid*" column. Of course, this information is not available to our disassembler. An example for the overlap between valid and invalid instructions can be seen between the second and the third instruction. The valid instruction at address `0x8048001` requires two bytes and thus interferes with the next (invalid) instruction at `0x8048002`.

				Valid	*Candidate*
8048000	55	push	%ebp	x	
8048001	89 e5	mov	%esp, %ebp	x	
8048002	e5 e8	in	e8,%eax		
8048003	e8 00 00 74 11	call	19788008 <obfuscator>	x	
8048004	00 00	add	%al, %eax		
8048005	00 74	add			
8048006	74 11	je	8048019		x
...					
804800c	75 06	jne	8048014	x	x
...					
8048010	eb 07	jmp	8048019	x	x
...					
8048017	74 01	je	804801a		x
8048018	01 89 ec 5d c3 90	add	%dh,ffffff89(%ecx,%eax,1)		
8048019	89 ec	mov	%ebp, %esp	x	
804801a	ec	in	(%dx), %al		
804801b	5d	pop	%ebp	x	
...					

Fig. 1.2. Partial instruction listing.

The next step is to identify all intra-procedural control transfer instructions. For our purposes, an intra-procedural control transfer instruction is defined as a CTI with at least one known successor basic block in the same function. Remember that we assume that control flow only continues after conditional branches but not necessarily after call or unconditional branch instructions. Therefore, an instruction is an intra-procedural control transfer instruction if either (**i**) its target address can be determined and this address is in the range between the function's start and end addresses or (**ii**) it is a conditional jump. In the latter case, the address that immediately follows the conditional jump instruction is the start of a successor block.

Note that we assume that a function is represented by a contiguous sequence of instructions, with possible junk instructions added in between. This means that, it is not possible that the basic blocks of two different functions are intertwined. Therefore, each function has one start address and one end address (i.e., the last instruction of the last basic block that belongs to this function). However, it is possible that a function has multiple exit points.

To find all intra-procedural CTIs, the instructions decoded in the previous step are scanned for any control transfer instructions. For each CTI found in this way, we attempt to extract its target address. In the current implementation, only direct address modes are supported and no data flow analysis is performed to compute address values used by indirect jumps. However, such analysis could be later added to further improve the performance of our static analyzer. When the instruction is determined to be an intra-procedural control transfer operation, it is included in the set of *jump candidates*. The jump candidates of the sample function are marked in Figure 1.2 by an `x` in the

"*Candidate*" column. In this example, the call at address `0x8048003` is not included into the set of jump candidates because the target address is located outside the function.

Given the set of jump candidates, an initial control flow graph is constructed. This is done with the help of a recursive disassembler. Starting with an initial empty CFG, the disassembler is successively invoked for all the elements in the set of jump candidates. In addition, it is also invoked for the instruction at the start address of the function.

The key idea for taking into account all possible control transfer instructions is the fact that the valid CTIs determine the skeleton of the analyzed function. By using *all* control flow instructions to create the initial CFG, we make sure that the real CFG is a subgraph of this initial graph. Because the set of jump candidates can contain both valid and invalid instructions, it is possible (and also frequent) that the initial CFG contains a superset of the nodes of the real CFG. These nodes are introduced as a result of argument bytes of valid instructions being misinterpreted as control transfer instructions. The Intel x86 instruction set contains 26 single-byte opcodes that map to control transfer instructions (out of 219 single-byte instruction opcodes). Therefore, the probability that a random argument byte is decoded as CTI is not negligible. In our experiments [11], we found that about one tenth of all decoded instructions are CTIs. Of those instructions, only two thirds were part of the real control flow graph. As a result, the initial CFG contains nodes and edges that represent invalid instructions. Most of the time, these nodes contain instructions that overlap with valid instructions of nodes that belong to the real CFG. The following section discusses mechanisms to remove these spurious nodes from the initial control flow graph. It is possible to distinguish spurious from valid nodes because invalid CTIs represent random jumps within the function while valid CTIs constitute a well-structured CFG with nodes that have no overlapping instructions.

Creating an initial CFG that includes nodes that are not part of the real control flow graph can been seen as the opposite to the operation of a recursive disassembler. A standard recursive disassembler starts from a known valid block and builds up the CFG by adding nodes as it follows the targets of control transfer instructions that are encountered. This technique seems favorable at a first glance, because it makes sure that no invalid instructions are incorporated into the CFG. However, most control flow graphs are partitioned into several unconnected subgraphs. This happens because there are control flow instructions such as indirect branches whose targets often cannot be determined statically. This leads to missing edges in the CFG and to the problem that only a fraction of the real control flow graph is reachable from a certain node. The situation is exacerbated when dealing with obfuscated binaries, as inter-procedural calls and jumps are redirected to a branching function that uses indirect jumps. This significantly reduces the parts of the control flow graph that are directly accessible to a recursive disassembler, leading to unsatisfactory results.

Although the standard recursive disassembler produces suboptimal results, we use a similar algorithm to extract the basic blocks to create the initial CFG. As mentioned before, however, the recursive disassembler is not only invoked for the start address of the function alone, but also for all jump candidates that have been identified. An initial control flow graph is then constructed.

There are two differences between a standard recursive disassembler and our prototype tool. First, we assume that the address after a call or an unconditional jump instruction does not have to contain a valid instruction. Therefore, our recursive disassembler cannot continue at the address following a call or an unconditional jump. Note, however, that we do continue to disassemble after a conditional jump (i.e., branch).

The second difference is due to the fact that it is possible to have instructions in the initial call graph that overlap. In this case, two different basic blocks in the call graph can contain overlapping instructions starting at slightly different addresses. When following a sequence of instructions, the disassembler can arrive at an instruction that is already part of a previously found basic block. Normally, this instruction is the first instruction of the existing block. The disassembler can then "close" the instruction sequence of the current block and create a link to the existing basic block in the control flow graph.

When instructions can overlap, it is possible that the current instruction sequence overlaps with another sequence in an existing basic block for some instructions before the two sequences eventually become identical. In this case, the existing basic block is split into two new blocks. One block refers to the overlapping sequence up to the instruction where the two sequences merge, the other refers to the instruction sequence that both have in common. All edges in the control flow graph that point to the original basic block are changed to point to the first block, while all outgoing edges of the original block are assigned to the second. In addition, the first block is connected to the second one.

The reason for splitting the existing block is the fact that a basic block is defined as a continuous sequence of instructions without a jump or jump target in the middle. When two different overlapping sequences merge at a certain instruction, this instruction has two predecessor instructions (one in each of the two overlapping sequences). Therefore, it becomes the first instruction of a new basic block. As an additional desirable side effect, each instruction appears at most once in a basic block of the call graph.

The fact that instruction sequences eventually "merge" is a common phenomenon when disassembling x86 binaries. The reason is called *self-repairing disassembly* and relates to the fact that two instruction sequences that start at slightly different addresses (that is, shifted by a few bytes) synchronize quickly, often after a few instructions. Therefore, when the disassembler starts at an address that does not correspond to a valid instruction, it can be expected to re-synchronize with the sequence of valid instructions after a few steps [13].

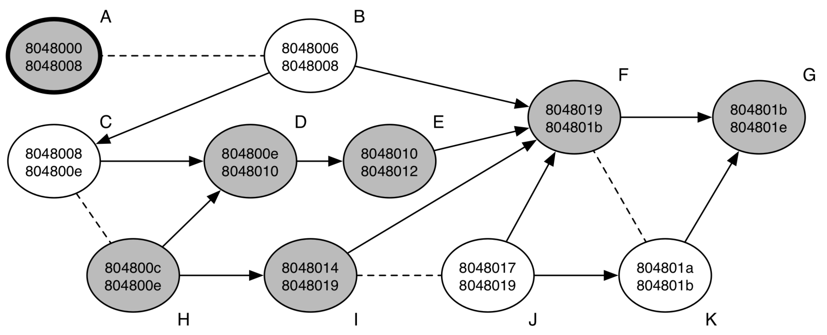


Fig. 1.3. Initial control flow graph.

The initial control flow graph generated for for our example function is shown in Figure 1.3. In this example, the algorithm is invoked for the function start at address `0x8048000` and the four jump candidates (`0x8048006`, `0x804800c`, `0x8048010`, and `0x8048017`). The nodes in this figure represent basic blocks and are labeled with the start address of the first instruction and the end address of the last instruction in the corresponding instruction sequence. Note that the end address denotes the first byte *after* the last instruction and is not part of the basic block itself. Solid, directed edges between nodes represent the targets of control transfer instructions. A dashed line between two nodes signifies a *conflict* between the two corresponding blocks.

Two basic blocks are in conflict when they contain at least one pair of instructions that overlap. As discussed previously, our algorithm guarantees that a certain instruction is assigned to at most one basic block (otherwise, blocks are split appropriately). Therefore, whenever the address ranges of two blocks overlap, they must also contain different, overlapping instructions. Otherwise, both blocks would contain the same instruction, which is not possible. This is apparent in Figure 1.3, where the address ranges of all pairs of conflicting basic blocks overlap. To simplify the following discussion of the techniques used to resolve conflicts, nodes that belong to the real control flow graph are shaded. In addition, each node is denoted with an uppercase letter.

1.2.4 Block Conflict Resolution

The task of the block conflict resolution phase is to remove basic blocks from the initial CFG until no conflicts are present anymore. Conflict resolution proceeds in five steps. The first two steps remove blocks that are *definitely* invalid, given our assumptions. The last three steps are heuristics that choose *likely* invalid blocks. The conflict resolution phase terminates immediately after the last conflicting block is removed; it is not necessary to carry out all steps. The final step brings about a decision for any basic block conflict and the control flow graph is guaranteed to be free of any conflicts when the conflict resolution phase completes.

The five steps are detailed in the following paragraphs.

Step 1: We assume that the start address of the analyzed function contains a valid instruction. Therefore, the basic block that contains this instruction is valid. In addition, whenever a basic block is known to be valid, all blocks that are reachable from this block are also valid.

A basic block v is *reachable* from basic block u if there exists a path p from u to v. A path p from u to v is defined as a sequence of edges that begins at u and terminates at v. An edge is inserted into the control flow graph only when its target can be statically determined and a possible program execution trace exists that transfers control over this edge. Therefore, whenever a control transfer instruction is valid, its targets have to be valid as well.

We tag the node that contains the instruction at the function's start address and all nodes that are reachable from this node as valid. Note that this set of valid nodes contains exactly the nodes that a traditional recursive disassembler would identify when invoked with the function's start address. When the valid nodes are identified, any node that is in conflict with at least one of the valid nodes can be removed.

In the initial control flow graph for the example function in Figure 1.3, only node A (`0x8048000`) is marked as valid. That node is drawn with a stronger border in Figure 1.3. The reason is that the corresponding basic block ends with a call instruction at `0x8048003` whose target is not local. In addition, we do not assume that control flow resumes at the address after a call and thus the analysis cannot directly continue after the call instruction. In Figure 1.3, node B (the basic block at `0x8048006`) is in conflict with the valid node and can be removed.

Step 2: Because of the assumption that valid instructions do not overlap, it is not possible to start from a valid block and reach two different nodes in the control flow graph that are in conflict. That is, whenever two conflicting nodes are both reachable from a third node, this third node cannot be valid and is removed from the CFG. The situation can be restated using the notion of a common ancestor node. A common ancestor node of two nodes u and v is defined as a node n such that both u and v are reachable from n.

In Step 2, all common ancestor nodes of conflicting nodes are removed from the control flow graph. In our example in Figure 1.3, it can be seen that the conflicting node F and node K share a common ancestor, namely node J. This node is removed from the CFG, resolving a conflict with node I. The resulting control flow graph after the first two steps is shown in Figure 1.4.

The situation of having a common ancestor node of two conflicting blocks is frequent when dealing with invalid conditional branches. In such cases, the branch target and the continuation after the branch instruction are often directly in conflict, allowing one to remove the invalid basic block from the control flow graph.

Step 3: When two basic blocks are in conflict, it is reasonable to expect that a valid block is more tightly integrated into the control flow graph than a block that was created because of a misinterpreted argument value of a

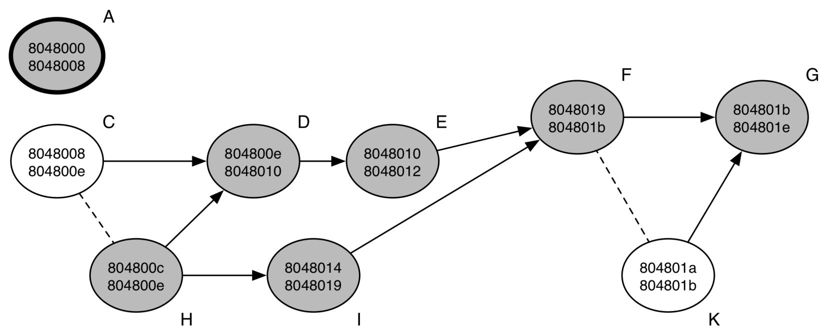


Fig. 1.4. CFG after two steps of conflict resolution.

program instruction. That means that a valid block is often reachable from a substantial number of other blocks throughout the function, while an invalid block usually has only a few ancestors.

The degree of integration of a certain basic block into the control flow graph is approximated by the number of its predecessor nodes. A node u is defined as a *predecessor node* of v when v is reachable from u. In Step 3, the predecessor nodes for pairs of conflicting nodes are determined and the node with the smaller number is removed from the CFG.

In Figure 1.4, node K has no predecessor nodes while node F has five. Note that the algorithm cannot distinguish between real and spurious nodes and, thus, it includes node C in the set of predecessor nodes for node F. As a result, node K is removed. The number of predecessor nodes for node C and node H are both zero and no decision is made in the current step.

Step 4: In this step, the number of direct successor nodes of two conflicting nodes are compared. A node v is a *direct successor node* of node u when v can be directly reached through an outgoing edge from u. The node with less direct successor nodes is then removed. The rationale behind preferring the node with more outgoing edges is the fact that each edge represents a jump target within the function and it is more likely that a valid control transfer instruction has a target within the function than any random CTI.

In Figure 1.4, node C has only one direct successor node while node H has two. Therefore, node C is removed from the control flow graph. In our example, all conflicts are resolved at this point.

Step 5: In this step, all conflicts between basic blocks must be resolved. For each pair of conflicting blocks, one is chosen at random and then removed from the graph. No human intervention is required at this step, but it would be possible to create different alternative disassembly outputs (one output for each block that needs to be removed) that can be all presented to a human analyst.

It might also be possible to use statistical methods during Step 5 to improve the chances that the "correct" block is selected. However, this technique is not implemented and is left for future work.

The result of the conflict resolution step is a control flow graph that contains no overlapping basic blocks. The instructions in these blocks are considered valid and could serve as the output of the static analysis process. However, most control flow graphs do not cover the function's complete address range and gaps exist between some basic blocks.

1.2.5 Gap Completion

The task of the gap completion phase is to improve the results of our analysis by filling the gaps between basic blocks in the control flow graph with instructions that are likely to be valid. A *gap* from basic block b_1 to basic block b_2 is the sequence of addresses that starts at the first address after the end of basic block b_1 and ends at the last address before the start of block b_2, given that there is no other basic block in the control flow graph that covers any of these addresses. In other words, a gap contains bytes that are not used by any instruction in blocks the control flow graph.

Gaps are often the result of junk bytes that are inserted by the obfuscator. Because junk bytes are not reachable at run-time, the control flow graph does not cover such bytes. It is apparent that the attempt to disassemble gaps filled with junk bytes does not improve the results of the analysis. However, there are also gaps that do contain valid instructions. These gaps can be the result of an incomplete control flow graph, for example, stemming from a region of code that is only reachable through an indirect jump whose target cannot be determined statically. Another frequent cause for gaps that contain valid instructions are call instructions. Because the disassembler cannot continue after a call instruction, the following valid instructions are not immediately reachable. Some of these instructions might be included into the control flow graph because they are the target of other control transfer instructions. Those regions that are not reachable, however, cause gaps that must be analyzed in the gap completion phase.

The algorithm to identify the most probable instruction sequence in a gap from basic block b_1 to basic block b_2 works as follows. First, all possibly valid sequences in the gap are identified. A necessary condition for a valid instruction sequence is that its last instruction either (**i**) ends with the last byte of the gap or (**ii**) its last instruction is a non intra-procedural control transfer instruction. The first condition states that the last instruction of a valid sequence has to be directly adjacent to the first instruction of block b_2. This becomes evident when considering a valid instruction sequence in the gap that is executed at run-time. After the last instruction of the sequence is executed, the control flow has to continue at the first instruction of basic block b_2. The second condition states that a sequence does not need to end directly adjacent to block b_2 if the last instruction is a non intra-procedural control transfer. The restriction to non intra-procedural CTIs is necessary because all intra-procedural CTIs are included into the initial control flow graph. When an intra-procedural instruction appears in a gap, it must have been removed during the conflict resolution phase and should not be included again.

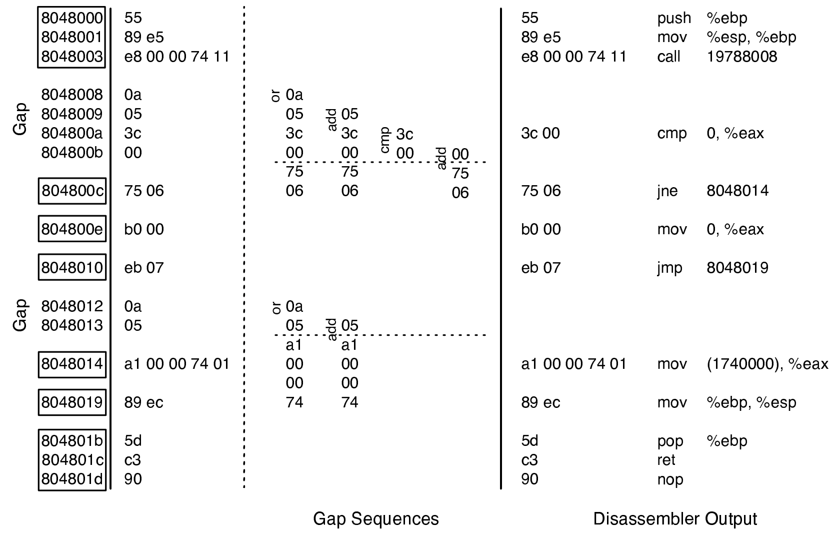
Instruction sequences are found by considering each byte between the start and the end of the gap as a potential start of a valid instruction sequence. Subsequent instructions are then decoded until the instruction sequence either meets or violates one of the necessary conditions defined above. When an instruction sequence meets a necessary condition, it is considered possibly valid and a *sequence score* is calculated for it. The sequence score is a measure of the likelihood that this instruction sequence appears in an executable. It is calculated as the sum of the *instruction scores* of all instructions in the sequence. The instruction score is similar to the sequence score and reflects the likelihood of an individual instruction. Instruction scores are always greater or equal than zero. Therefore, the score of a sequence cannot decrease when more instructions are added. We calculate instruction scores using statistical techniques and heuristics to identify improbable instructions.

The statistical techniques are based on instruction probabilities and digraphs. Our approach utilizes tables that denote both the likelihood of individual instructions appearing in a binary as well as the likelihood of two instructions occurring as a consecutive pair. The tables were built by disassembling a large set of common executables and tabulating counts for the occurrence of each individual instruction as well as counts for each occurrence of a pair of instructions. These counts were subsequently stored for later use during the disassembly of an obfuscated binary. It is important to note that only instruction opcodes are taken into account with this technique; operands are not considered. The basic score for a particular instruction is calculated as the sum of the probability of occurrence of this instruction and the probability of occurrence of this instruction followed by the next instruction in the sequence.

In addition to the statistical technique, a set of heuristics is used to identify improbable instructions. This analysis focuses on instruction arguments and observed notions of the validity of certain combinations of operations, registers, and accessing modes. Each heuristic is applied to an individual instruction and can modify the basic score calculated by the statistical technique. In our current implementation, the score of the corresponding instruction is set to zero whenever a rule matches. Examples of these rules include the following:

- operand size mismatches;
- certain arithmetic on special-purpose registers;
- unexpected register-to-register moves (e.g., moving from a register other than %ebp into %esp);
- moves of a register value into memory referenced by the same register.

When all possible instruction sequences are determined, the one with the highest sequence score is selected as the valid instruction sequence between b_1 and b_2.



```
8048000  55                  55                push  %ebp
8048001  89 e5               89 e5             mov   %esp, %ebp
8048003  e8 00 00 74 11      e8 00 00 74 11    call  19788008
8048008  0a
8048009  05
804800a  3c                  3c 00             cmp   0, %eax
804800b  00
804800c  75 06               75 06             jne   8048014
804800e  b0 00               b0 00             mov   0, %eax
8048010  eb 07               eb 07             jmp   8048019
8048012  0a
8048013  05
8048014  a1 00 00 74 01      a1 00 00 74 01    mov   (1740000), %eax
8048019  89 ec               89 ec             mov   %ebp, %esp
804801b  5d                  5d                pop   %ebp
804801c  c3                  c3                ret
804801d  90                  90                nop
```

Fig. 1.5. Gap completion and disassembler output.

The instructions that make up the control flow graph of our example function and the intermediate gaps are shown in the left part of Figure 1.5. It can be seen that only a single instruction sequence is valid in the first gap, while there is none in the second gap. The right part of Figure 1.5 shows the output of our disassembler. All valid instructions of the example function have been correctly identified.

Based on the list of valid instructions, the subsequent code analysis phase can attempt to detect malicious code. In the following Section 1.3, we present symbolic execution as one possible static analysis approach to identify higher-level properties of code.

1.3 Code Analysis

This section describes the use of symbolic execution [10], a static analysis technique to identify code sequences that exhibit certain properties. In particular, we aim at characterizing a code piece by its semantics, or, in other words, by its effect on the environment. The goal is to construct models that characterize malicious behavior, regardless of the particular sequence of instructions (and therefore, of bytes) used in the code. This allows one to specify more general and robust descriptions of malicious code that cannot be evaded by simple changes to the syntactic representation or layout of the code (e.g., by renaming registers or modify the execution order of instructions).

Symbolic execution is a technique that interpretatively executes a program, using symbolic expressions instead of real values as input. This also includes the execution environment of the program (data, stack, and heap regions) for which no initial value is known at the time of the analysis. Of course, for all variables for which concrete values are known (e.g., initialized data segments), these values are used. When the execution starts from the entry point in the program, say address s, a symbolic execution engine interprets the sequence of machine instructions as they are encountered in the program.

To perform symbolic execution of machine instructions (in our case, Intel x86 operations), it is necessary to extend the semantics of these instructions so that operands are not limited to real data objects but can also be symbolic expressions. The normal execution semantics of Intel x86 assembly code describes how data objects are represented, how statements and operations manipulate these data objects, and how control flows through the statements of a program. For symbolic execution, the definitions for the basic operators of the language have to be extended to accept symbolic operands and produce symbolic formulas as output.

1.3.1 Execution State

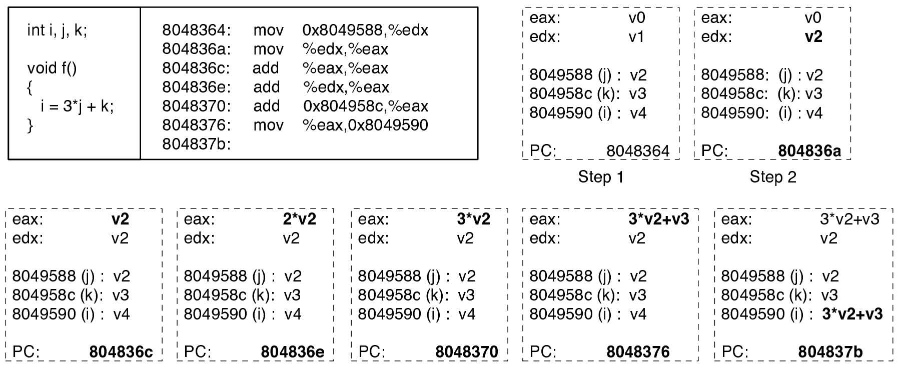
We define the execution state S of program p as a snapshot of the content of the processor registers (except the program counter) and all valid memory locations at a particular instruction of p, which is denoted by the program counter. Although it would be possible to treat the program counter like any other register, it is more intuitive to handle the program counter separately and to require that it contain a concrete value (i.e., it points to a certain instruction). The content of all other registers and memory locations can be described by symbolic expressions.

Before symbolic execution starts from address s, the execution state S is initialized by assigning symbolic variables to all processor registers (except the program counter) and memory locations for which no concrete value is known initially. Thus, whenever a processor register or a memory location is read for the first time, without any previous assignment to it, a new symbol is supplied from the list of variables $\{\upsilon_l, \upsilon_2, \upsilon_3, \ldots\}$. Note that this is the only time when symbolic data objects are introduced.

In our current system, we do not support floating-point data objects and operations. Therefore, all symbols (variables) represent integer values. Symbolic expressions are linear combinations of these symbols (i.e., integer polynomials over the symbols). A symbolic expression can be written as $c_n * \upsilon_n + c_{n-1} * \upsilon_{n-1} + \ldots + c_1 * \upsilon_1 + c_0$ where the c_i are constants. In addition, there is a special symbol $\perp$ that denotes that no information is known about the content of a register or a memory location. Note that this is very different from a symbolic expression. Although there is no *concrete* value known for a symbolic expression, its value can be evaluated when concrete values are supplied for the initial execution state. For the symbol $\perp$, nothing can be asserted, even when the initial state is completely defined.

By allowing program variables to assume integer polynomials over the symbols v_i, the symbolic execution of assignment statements follows naturally. The expression on the right-hand side of the statement is evaluated, substituting symbolic expressions for source registers or memory locations. The result is another symbolic expression (an integer is the trivial case) that represents the new value of the left-hand side of the assignment statement. Because symbolic expressions are integer polynomials, it is possible to evaluate addition and subtraction of two arbitrary expressions. Also, it is possible to multiply or shift a symbolic expression by a constant value. Other instructions, such as the multiplication of two symbolic variables or a logic operation (e.g., `and`, `or`), result in the assignment of the symbol $\perp$ to the destination. This is because the result of these operations cannot (always) be represented as integer polynomial. The reason for limiting symbolic formulas to linear expressions will become clear in Section 1.3.3.

Whenever an instruction is executed, the execution state is changed. As mentioned previously, in case of an assignment, the content of the destination operand is replaced with the right-hand side of the statement. In addition, the program counter is advanced. In the case of an instruction that does not change the control flow of a program (i.e., an instruction that is not a jump or a conditional branch), the program counter is simply advanced to the next instruction. Also, an unconditional jump to a certain label (instruction) is performed exactly as in normal execution by transferring control from the current statement to the statement associated with the corresponding label.



```
int i, j, k;

void f()
{
  i = 3*j + k;
}
```

```
8048364:   mov   0x8049588,%edx
804836a:   mov   %edx,%eax
804836c:   add   %eax,%eax
804836e:   add   %edx,%eax
8048370:   add   0x804958c,%eax
8048376:   mov   %eax,0x8049590
804837b:
```

```
Step 1
eax:              v0
edx:              v1

8049588 (j) :  v2
804958c (k):  v3
8049590 (i) :  v4

PC:         8048364
```

```
Step 2
eax:              v0
edx:              v2

8049588:  (j) : v2
804958c:  (k): v3
8049590:  (i) : v4

PC:         804836a
```

```
Step 3
eax:              v2
edx:              v2

8049588 (j) :  v2
804958c (k):  v3
8049590 (i) :  v4

PC:         804836c
```

```
Step 4
eax:            2*v2
edx:              v2

8049588 (j) :  v2
804958c (k):  v3
8049590 (i) :  v4

PC:         804836e
```

```
Step 5
eax:            3*v2
edx:              v2

8049588 (j) :  v2
804958c (k):  v3
8049590 (i) :  v4

PC:         8048370
```

```
Step 6
eax:         3*v2+v3
edx:              v2

8049588 (j) :  v2
804958c (k):  v3
8049590 (i) :  v4

PC:         8048376
```

```
Step 7
eax:         3*v2+v3
edx:              v2

8049588 (j) :  v2
804958c (k):  v3
8049590 (i) :  3*v2+v3

PC:         804837b
```

Fig. 1.6. Symbolic execution.

Figure 1.6 shows the symbolic execution of a sequence of instructions. In addition to the x86 machine instructions, a corresponding fragment of C source code is shown. For each step of the symbolic execution, the relevant parts of the execution state are presented. Changes between execution states are shown in bold face. Note that the compiler (`gcc 3.3`) converted the multiplication in the C program into an equivalent series of `add` machine instructions.

1.3.2 Conditional Branches and Loops

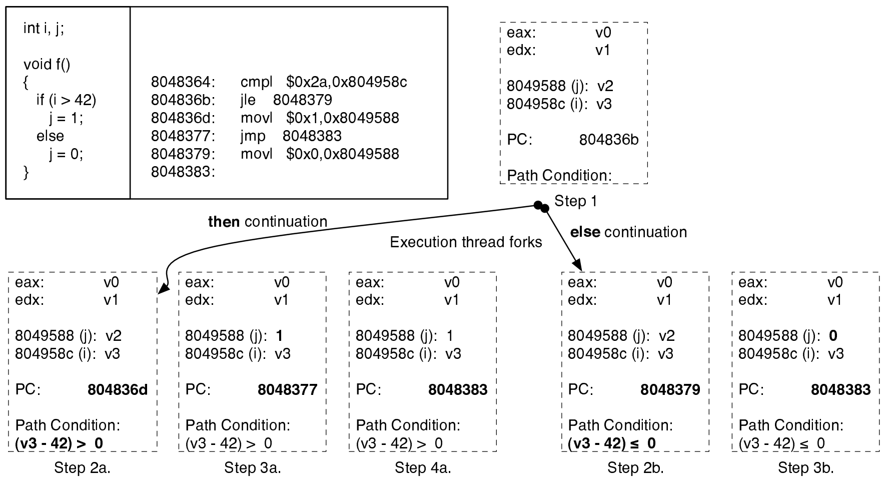
To handle conditional branches, the execution state has to be extended to include a set of constraints, called the *path constraints*. In principle, a path constraint relates a symbolic expression L to a constant. This can be used, for example, to specify that the content of a register has to be equal to 0. More formally, a path constraint is a boolean expression of the form $L \geq 0$ or $L = 0$, in which L is an integer polynomial over the symbols υ_i. The set of path constraints forms a linear constraint system.

The symbolic execution of a conditional branch statement starts by evaluating the associated Boolean expression. The evaluation is done by replacing the instruction's operands with their corresponding symbolic expressions. Then, the inequality (or equality) is transformed and converted into the standard form introduced above. Let the resulting path constraint be called q.

To continue symbolic execution, both branches of the control path need to be explored. The symbolic execution forks into two "parallel" execution threads: one thread follows the *then* alternative, while the other one follows the *else* alternative. Both execution threads assume the execution state that existed immediately before the conditional statement, but proceed independently thereafter. Because the *then* alternative is only chosen if the conditional branch is taken, the corresponding path constraint q must be true. Therefore, we add q to the set of path constraints of this execution thread. The situation is reversed for the *else* alternative. In this case, the branch is not taken and q must be false. Thus, $\neg q$ is added to the path constraints of this execution.

After q (or $\neg q$) is added to a set of path constraints, the corresponding linear constraint system is immediately checked for satisfiability. When the set of path constraints has no solution, this implies that, independent of the choice of values for the initial configuration C, this path of execution can never occur. This allows us to immediately terminate impossible execution threads.

Each fork of execution at a conditional statement contributes a condition over the variables υ_i that must hold for this particular execution thread. Thus, the set of path constraints determines which conditions the initial execution state must satisfy in order for an execution to follow the particular associated path. Each symbolic execution begins with an empty set of path constraints. As assumptions about the variables are made (in order to choose between alternative paths through the program as presented by conditional statements), those assumptions are added to the set. An example of a fork into two symbolic execution threads as the result of an `if`-statement and the



```
int i, j;

void f()
{
   if (i > 42)
      j = 1;
   else
      j = 0;
}
```

```
8048364:    cmpl   $0x2a,0x804958c
804836b:    jle    8048379
804836d:    movl   $0x1,0x8049588
8048377:    jmp    8048383
8048379:    movl   $0x0,0x8049588
8048383:
```

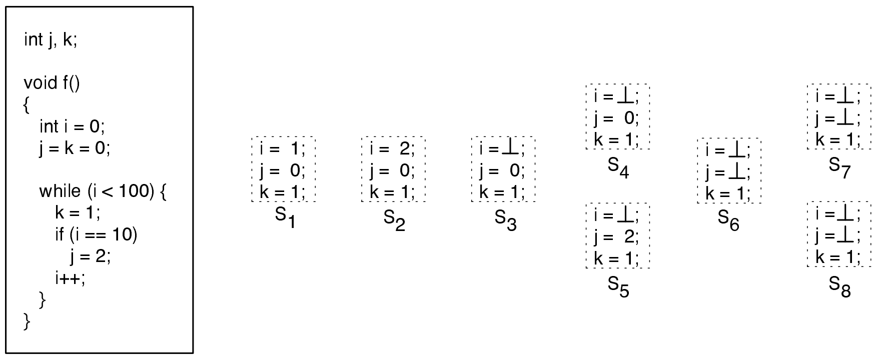
Fig. 1.7. Handling conditional branches during symbolic execution.

corresponding path constraints are shown in Figure 1.7. Note that the `if`-statement was translated into two machine instructions. Thus, special code is required to extract the condition on which a branch statement depends.

Because a symbolic execution thread forks into two threads at each conditional branch statement, loops represent a problem. In particular, we have to make sure that execution threads "make progress." The problem is addressed by requiring that a thread passes through the same loop at most three times. Before an execution thread enters a loop for the forth time, its execution is halted. Then, the effect of an arbitrary number of iterations of this loop on the execution state is approximated. This approximation is a standard static analysis technique [6, 14] that aims at determining value ranges for the variables that are modified in the loop body. Since the problem of finding exact ranges and relationships between variables is undecidable in the general case, the approximation naturally involves a certain loss of precision. After the effect of the loop on the execution thread is approximated, the thread can continue with the modified state after the loop. To determine loops in the control flow graph, we use the algorithm by Lengauer-Tarjan [12], which is based on dominator trees.

To approximate the effect of the loop body on an execution state, a *fixpoint* for this loop is constructed. For our purposes, a fixpoint is an execution state F that, when used as the initial state before entering the loop, is equivalent to the execution state after the loop termination. In other words, after the operations of the loop body are applied to the fixpoint state F, the resulting execution state is again F. Clearly, if there are multiple paths through the loop, the resulting execution states at each loop exit must be the same (and identical to F). Thus, whenever the effect of a loop on an execution state must be determined, we transform this state into a fixpoint for this loop. This transformation is often called *widening*. Then, the thread can continue after the loop using the fixpoint as its new execution state.

The fixpoint for a loop is constructed in an iterative fashion. Given the execution state S_1 after the first execution of the loop body, we calculate the execution state S_2 after a second iteration. Then, S_1 and S_2 are compared. For each register and each memory location that hold different values (i.e., different symbolic expressions), we assign $\perp$ as the new value. The resulting state is used as the new state and another iteration of the loop is performed. This is repeated until S_i and $S_{(i+1)}$ are identical. In case of multiple paths through the loop, the algorithm is extended by collecting one exit state S_i for each path and then comparing all pairs of states. Whenever a difference between a register value or a memory location is found, this location is set to $\perp$. The iterative algorithm is guaranteed to terminate, because at each step, it is only possible to convert the content of a memory location or a register to $\perp$. Thus, after each iteration, the states are either identical or the content of some locations is made unknown. This process can only be repeated until all values are converted to unknown and no information is left.



```
int j, k;

void f()
{
  int i = 0;
  j = k = 0;

  while (i < 100) {
    k = 1;
    if (i == 10)
      j = 2;
    i++;
  }
}
```

Fig. 1.8. Fixpoint calculation.

An example for a fixpoint calculation (using C code instead of x86 assembly) is presented in Figure 1.8. In this case, the execution state includes the values of the three variables i, j, and k. After the first loop iteration, the execution state S_1 is reached. Here, i has been incremented once, k has been assigned the constant 1, and j has not been modified. After a second iteration, S_2 is reached. Because i has changed between S_1 and S_2, its value is set to $\perp$ in S_3. Note that the execution has not modified j, because the value of i was known to be different from 10 at the `if`-statement. Using S_3 as the new execution state, two paths are taken through the loop. In one case (S_4), j is set to 2, in the other case (S_5), the variable j remains 0. The reason for the two different execution paths is the fact that i is no longer known at the `if`-statement and, thus, both paths have to be followed. Comparing S_3 with

S_4 and S_5, the difference between the values of variable j leads to the new state S_6 in which j is set to $\perp$. As before, the new state S_6 is used for the next loop iteration. Finally, the resulting states S_7 and S_8 are identical to S_6, indicating that a fixpoint is reached.

In the example above, we quickly reach a fixpoint. In general, by considering all modified values as unknown (setting them to $\perp$), the termination of the fixpoint algorithm is achieved very quickly. However, the approximation might be unnecessarily imprecise. For our current prototype, we use this simple approximation technique [14]. However, we plan to investigate more sophisticated fixpoint algorithms in the future.

1.3.3 Analyzing Effects of Code Sequences

As mentioned previously, the aim of the symbolic execution is to characterize the behavior of a piece of code. For example, symbolic execution could be used to determine if a system call is invoked with a particular argument. Another example is the assignment of a value to a certain memory address.

Consider a specification that defines a piece of code as malicious when it writes to an area in memory that should not be modified. Such a specification can be used to characterize kernel-level rootkits, which modify parts of the operating system memory (such as the system call table) that benign modules do not touch. To determine whether a piece of code can assign a value to a certain memory address t, the destination addresses of data transfer instructions (e.g., x86 `mov`) must be determined. Thus, whenever the symbolic execution engine encounters such an instruction, it checks whether this instruction can possibly access (or write to) address t. To this end, the symbolic expression that represents the destination of the data transfer instruction is analyzed. The reason is that if it were possible to force this symbolic expression to evaluate to t, then the attacker could achieve her goal.

Let the symbolic expression of the destination of the data transfer instruction be called s_t. To check whether it is possible to force the destination address of this instruction to t, the constraint $s_t = t$ is generated (this constraint simply expresses the fact that s_t should evaluate to the target address t). Now, we have to determine whether this constraint can be satisfied, given the current path constraints. To this end, the constraint $s_t = t$ is added to the path constraints, and the resulting linear inequality system is solved.

If the linear inequality system has a solution, then the sequence of code instructions that were symbolically executed so far can possibly write to t. Note that, since the symbolic expressions are integer polynomials over variables that describe the *initial state* of the system, the solution to the linear inequality system directly provides concrete values for the initial configuration that will eventually lead to a value being written to t. For example, in the case of kernel-level rootkit detection, a kernel module would be classified as malicious if a data transfer instruction (in its initialization routine) can be used to modify the address t of an entry in the system call table.

To solve the linear constraint systems, we use the Parma Polyhedral Library (PPL) [1]. In general, solving a linear constraint system is exponential in the number of inequalities. However, the number of inequalities is usually small, and PPL uses a number of optimizations to reduce the resources required at run time.

1.3.4 Memory Aliasing and Unknown Stores

In the previous discussion, two problems were ignored that considerably complicate the analysis for real programs: memory aliasing and store operations to unknown destination addresses.

Memory aliasing refers to the problem that two different symbolic expressions s_1 and s_2 might point to the same address. That is, although s_1 and s_2 contain different variables, both expressions evaluate to the same value. In this case, the assignment of a value to an address that is specified by s_1 has unexpected side effects. In particular, such an assignment simultaneously changes the content of the location pointed to by s_2.

Memory aliasing is a typical problem in the static analysis of high-level languages with pointers (such as C). Unfortunately, the problem is exacerbated at the machine code level. The reason is that, in a high-level language, only a certain subset of variables can be accessed via pointers. Also, it is often possible to perform alias analysis that further reduces the set of variables that might be subject to aliasing. Thus, one can often guarantee that certain variables are not modified by write operations through pointers. At machine level, the address space is uniformly treated as an array of storage locations. Thus, a write operation could potentially modify any other variable.

In our prototype, we take an optimistic approach and assume that different symbolic expressions refer to different memory locations. This approach is motivated by the fact that most C compilers address local and global variables so that a distinct expression is used for each access to a different variable. In the case of global variables, the address of the variable is directly encoded in the instruction, making the identification of the variable particularly easy. For each local variable, the access is performed by calculating a different offset with respect to the value of the base pointer register (`%ebp`).

A store operation to an unknown address is related to the aliasing problem as such an operation could potentially modify any memory location. Here, one can choose one of two options. A conservative and safe approach must assume that any variable could have been overwritten and no information remains. The other approach assumes that such a store operation does not interfere with any variable that is part of the solution of the linear inequality system. While this leads to the possibility of false negatives, it significantly reduces the number of false positives.

1.4 Conclusions

The analysis of an unknown program requires that the binary is first disassembled into its corresponding assembly code representation. Based on the code instructions, static or dynamic code analysis techniques can then be used to classify the program as malicious or benign.

In this chapter, we have introduced a robust disassembler that produces good results even when the malicious code employs tricks to resists analysis. This is crucial for many security tools, including virus scanners [2] and intrusion detection systems [9].

We also introduced symbolic execution as one possible static analysis technique to infer semantic properties of code. This allows us to determine the effects of the execution of a piece of code. Based on this knowledge, we can construct general and robust models of malicious code. These models do not describe particular instances of malware, but capture the properties of a whole class of malicious code. Thus, it is more difficult for an attacker to evade detection by applying simple changes to the syntactic representation of the code.

References

1. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *9th International Symposium on Static Analysis*, 2002.
2. M. Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
3. C. Cifuentes and M. Van Emmerik. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 40(2-3), 2000.
4. C. Cifuentes and K. Gough. Decompilation of Binary Programs. *Software Practice & Experience*, 25(7):811–829, July 1995.
5. F. B. Cohen. Operating System Protection through Program Evolution. `http://all.net/books/IP/evolve.html`.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*, 1977.
7. Data Rescure. IDA Pro: Disassembler and Debugger. `http://www.datarescue.com/idabase/`, 2004.
8. Free Software Foundation. *GNU Binary Utilities*, Mar 2002. `http://www.gnu.org/software/binutils/manual/`.
9. J.T. Giffin, S. Jha, and B.P. Miller. Detecting manipulated remote call streams. In *In Proceedings of 11th USENIX Security Symposium*, 2002.
10. J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), 1976.
11. C. Kruegel, F. Valeur, W. Robertson, and G. Vigna. Static Analysis of Obfuscated Binaries. In *Usenix Security Symposium*, 2004.

12. T. Lengauer and R. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1), 1979.
13. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 290–299, Washington, DC, October 2003.
14. F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
15. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson. Binary Translation. *Digital Technical Journal*, 4(4), 1992.