Towards Static Analysis of Virtualization-Obfuscated Binaries

Johannes Kinder
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
johannes.kinder@epfl.ch

Abstract

Virtualization-obfuscation protects a program from manual or automated analysis by compiling it into bytecode for a randomized virtual architecture and attaching a corresponding interpreter. Static analysis appears to be helpless on such programs, where only the code of the interpreter is directly visible.

In this paper, we explain the particular challenges for statically analyzing the combination of interpreter and bytecode. Static analysis for computing possible variable values is commonly precise only to the program location. In the interpreter loop, however, this combines unrelated data flow information from different locations of the bytecode program.

To avoid this loss of information, we show how to lift an existing static analysis to an additional dimension of location, to become sensitive to the value of the virtual program counter. Thus, the static analysis merges data flow from equal bytecode locations only. We lift an existing analysis implemented in the JAKSTAB *static analyzer and present preliminary results for processing a virtualization-obfuscated binary.*

1. Introduction

Virtualization-obfuscation is a strong obfuscation scheme to harden programs against reverse engineering [9, 10, 1]. The underlying idea is to translate program code to instructions (bytecode) for a randomly generated virtual architecture and to synthesize a corresponding interpreter. This interpreter reproduces the observable behavior of the original program from the bytecode. Hence, an equivalent but protected version of the original program can be generated by using just the interpreter as code and storing the translated bytecode as data. To a reverse engineer or automated analyzer, the interpreter is immediately visible, but the bytecode, which determines the program semantics, is an incomprehensible data block.

While virtualization-obfuscation sees legitimate use for protecting key-validation schemes and license managers, it is particularly popular with malware authors. Due to architecture randomization, the bytecode and interpreter can vary greatly from one protected instance to the other, preventing the generation of reliable malware signatures. But the interpreter by itself is not malicious and could indeed be protecting a legitimate program; emitting a warning on just detecting the presence of the interpreter without knowledge about the bytecode semantics can lead to false positives.

In earlier work, we have argued for applying static analysis to x86 binaries, both for verifying specifications of API contracts [5] and to detect malware [7]. Static analysis can aid reverse engineering by extracting data flow information (invariants) about the program behavior. For instance, it can compute bounds for program variables or detect information flow between system calls. But when directly applied to a program protected by virtualization-obfuscation, it computes data flow information only over the interpreter. The invariants computed for locations in the interpreter cover many different instructions in the bytecode program. Thus, they are too weak to help reverse engineering and make automated methods such as malware detection or verification outright impossible. We address these challenges through the following contributions:

- We isolate and identify the effect of virtualization-obfuscation. Static analysis is unable to extract useful information due to effectively becoming location-insensitive, a phenomenon we call *domain flattening*: abstract states from different, unrelated locations of the original program are merged when analyzing the interpreter, requiring extremely conservative over-approximations.
- We show how to adapt an existing static analysis to overcome this problem. We can define a lifted

variant of the static analysis that keeps separate abstract states for each value of the virtual program counter (VPC). The lifted analysis then achieves the same precision on the obfuscated code as it would on the unobfuscated code.

- We lift *Bounded Address Tracking* [5], a static analysis specifically designed for analyzing binaries, to VPC-sensitivity and discuss the implementation in the JAKSTAB static analyzer. We present preliminary results for analyzing an obfuscated example executable.

2. Background

Using a running example of an interpreter and program, we now briefly introduce the concepts of virtualization obfuscation and static analysis.

2.1. Virtualization Obfuscation

Programs protected by virtualization-obfuscation all share the same general structure of looping over a large switch statement that distinguishes individual bytecode instructions. An example of a simple interpreter is shown in Figure 1. Note that the obfuscation usually targets binary code; we use a C-style high-level presentation here for ease of exposition. The original code translated to bytecode is stored within the static array `code`; all data the bytecode processes, including its stack, is allocated to a second array `data`. In each iteration of the loop, the interpreter reads an opcode from the location in the code array pointed to by the *virtual program counter* (VPC). Depending on the opcode, the interpreter then reads operand addresses from the code array and performs actions on locations in the data array. After interpreting an instruction, the VPC is set to point to the next instruction. For jumps, the VPC is incremented (conditionally) by the relative offset to the target address.

An obfuscation engine can generate many different bytecode representations for the same program by randomizing the virtual instruction set. One degree of randomness is the mapping from opcodes to bytecode instructions. But the engine can also merge or split bytecode instructions into larger or smaller functional units. Therefore, there is little hope of directly analyzing or pattern matching the bytecode to learn about the potential behavior of the program.

Currently, there are two main tools for virtualization obfuscation, Code Virtualizer (http://oreans.com/codevirtualizer.php) and VMProtect (http://vmpsoft.com). Both are advertised for protecting key-validation methods in shareware programs or license managers, but they are also widely used for encrypting malware. The architecture randomization provides enough entropy to make it impossible to precisely identify a piece of malware using the standard fingerprinting techniques employed in end-user anti-virus products. Often, anti-virus vendors choose to emit generic warnings on any program for which their heuristic components determine that it "looks" like it has been protected in this way—with all the potential for false positives.

```
int code = { ... };
int data = { ... };

void interpret() {
  int vpc = 0, op1, op2;
  while (true) {
    switch(code[vpc]) {
      case 03: // increment
        op1 = code[vpc + 1];
        data[op1]++;
        vpc += 2;
        break;
      case 08: // conditional jump
        op1 = code[vpc + 1];
        op2 = code[vpc + 2];
        if (data[op1] <= 0)
          vpc += data[op2]
        else
          vpc += 3;
        break;
      case 18: // call ext. function
        op1 = code[vpc + 1];
        apiCall(data[op1]);
        vpc += 2;
        break;
      case 52: // assignment
        op1 = code[vpc + 1];
        op2 = code[vpc + 2];
        data[op1] = data[op2];
        vpc += 3;
        break;
      default: // halt
        return;
    } // end switch
  } // end while
}
```

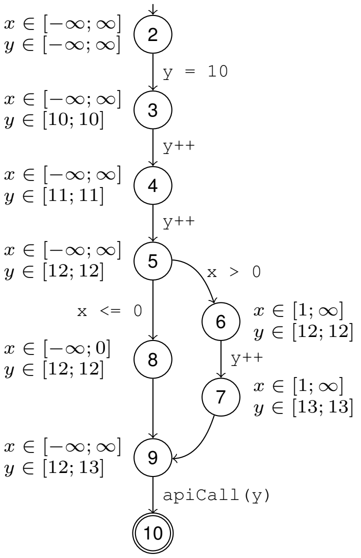
Figure 1. A simple interpreter.

2.2. Static Analysis

Static analysis reasons about a program without executing it. To ensure termination of the analysis, it introduces abstraction, i.e., it over-approximates the program semantics and analyzes a superset of all concrete behaviors of the program. This makes it dual to dynamic analysis, e.g., testing, which explores just a subset of the concrete behaviors by concretely executing the program on some inputs.

A static analysis is defined by an abstract domain

```
1  void foo(int x){
2    int y=10;
3    y++;
4    y++;
5    if (x > 0) {
6      y++;
7    }
8    else {}
9    apiCall(y);
10 }
```



$x \in [-\infty;\infty]$ $y \in [-\infty;\infty]$
2
y = 10
$x \in [-\infty;\infty]$ $y \in [10;10]$
3
y++
$x \in [-\infty;\infty]$ $y \in [11;11]$
4
y++
$x \in [-\infty;\infty]$ $y \in [12;12]$
5
x > 0
x <= 0
6
$x \in [1;\infty]$ $y \in [12;12]$
$x \in [-\infty;0]$ $y \in [12;12]$
8
y++
7
$x \in [1;\infty]$ $y \in [13;13]$
$x \in [-\infty;\infty]$ $y \in [12;13]$
9
apiCall(y)
10

Figure 2. Unobfuscated example program and its control flow graph, annotated with abstract states computed by a static interval analysis.

over which it operates (e.g., intervals) and an abstract transfer function that computes how the program execution affects the elements of the abstract domain (e.g., x++ changes the interval $x \in [0; 10]$ to $x \in [1; 11]$).

Consider the simple unobfuscated example program in Figure 2. An interval analysis is able to determine the (precise) bounds for variable y at the site of the external call in line 9. Starting from an initial state with infinite, unknown bounds for x and y, the transfer function for the assignment at line 2 computes a successor with a single-value interval of $[10; 10]$ for y. The analysis progresses through lines 3 and 4, updating the values as annotated in the control flow graph in Figure 2. From line 5, there are two outgoing control flow edges, one for each branch of the if statement. The then branch leads to the restricted interval $[1; \infty]$ for x, the increment at line 6 to $y \in [13; 13]$. The else branch restricts the interval for x to $[-\infty; 0]$. Both branches recombine in line 9. Because the analysis just keeps one interval per variable per location, it has to *merge* the intervals from both states. Therefore, x is again unbounded and y is known to be in the interval $[12; 13]$, which also constitutes the possible arguments to the external call in line 9.

For a program with loops, the analysis computes the transfer function repeatedly until it reaches a fixpoint over the map from locations to abstract states. To accelerate the finding of the fixpoint, which might otherwise require an infinite number of steps, the analysis can apply *widening* [2]. That is, it over-approximates the transfer function further to guarantee reaching a fixpoint in a finite number of steps. In the interval domain, widening typically involves setting one bound to infinity or intermediate program-specific constants.

Within the framework of abstract interpretation [2], a program analysis is often defined using a smaller abstract domain by lifting and pointwise extension of the transfer function. For instance, the full abstract domain for interval analysis can be constructed by lifting individual intervals of type $(\mathbb{N} \times \mathbb{N})$ for single variables to intervals for all variables $\mathbf{Var}$ at all program locations $\mathbf{L}$: interval analysis thus represents the data flow information for the program as a map from program locations to maps from variables to intervals, i.e., $\mathbf{L} \to \mathbf{Var} \to (\mathbb{N} \times \mathbb{N})$.

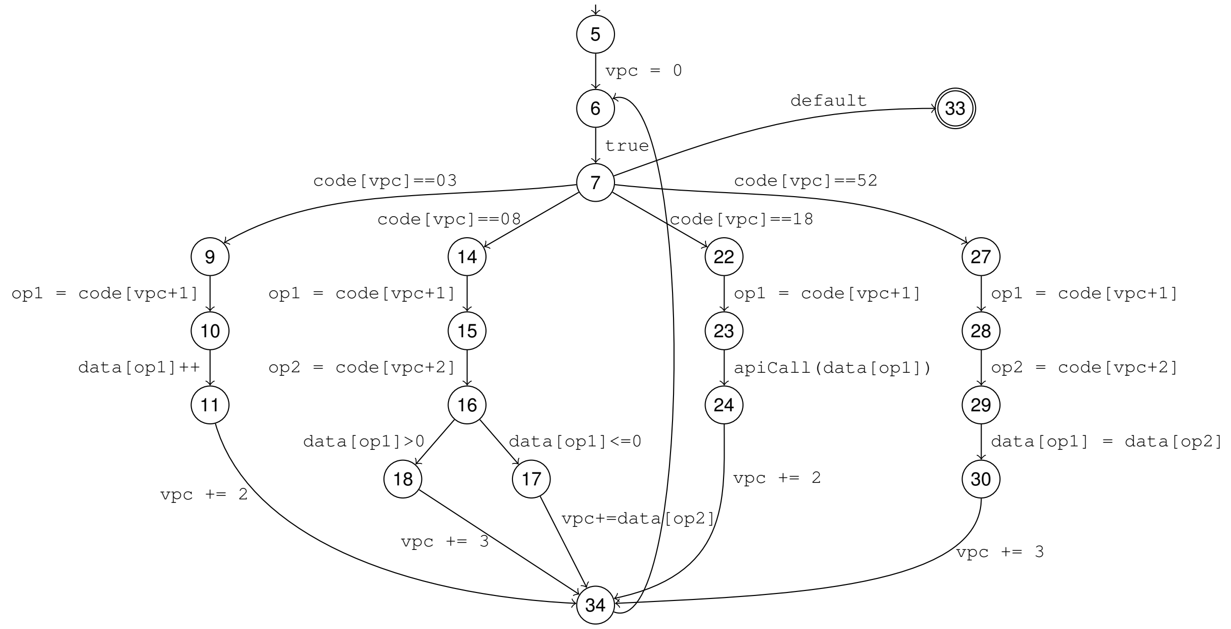
In the following we will only consider the common case of forward analyses that are lifted to individual program locations, i.e., their abstract domain is a mapping $\mathbf{D} = \mathbf{L} \to \mathbf{S}$ of locations to abstract states from the domain $\mathbf{S}$. We will refer to these analyses as being *location-sensitive*. For location-sensitive domains, it is convenient to define transfer functions for individual statements f_c of type $\mathbf{S} \times \mathbf{Stmt} \to \mathbf{S}$, which compute the successor of an abstract state with respect to a given statement c from the set of statements $\mathbf{Stmt}$. The transfer function $f :: \mathbf{D} \to \mathbf{D}$ for the full domain is then derived as

$$f(d) = \{(\ell_0 \mapsto \iota)\} \cup \left\{ \ell \mapsto \bigsqcup_{(\ell',C,\ell)\in G} f_c(d(\ell')) \,\middle|\, \ell \in \mathbf{L} \right\},$$

where ℓ_0 is the program entry point, ι is the initial abstract state, $G \subseteq \mathbf{L} \times \mathbf{Stmt} \times \mathbf{L}$ is the control flow graph, and $d(\ell')$ denotes s' such that $\ell' \mapsto s' \in d$. From an element d of $\mathbf{D}$, i.e., a map from locations to abstract states, f computes a new map by applying the per-statement transfer functions along each control flow edge. At control flow join points, the merging operator $\sqcup$ combines the abstract states from each incoming edge. Computing the analysis then amounts to finding the least fixpoint of f starting from the least (bottom) element of $\mathbf{D}$.

3. Static Analysis of Interpreted Code

When the simple program of Figure 2 is virtualized, it will appear to the static analyzer as the interpreter shown in Figure 1, with a control flow graph as shown in Figure 3. Everything specific to the original program is stored within the static code and data arrays. In some sense, the original program "is still there"; after all, executing the obfuscated version will produce the same observable behavior. But a static analyzer faces



code = { **52**, 01, 02, **03**, 01, **03**, 01, **08**, 00, 03, **03**, 01, **18**, 01, **00** }
data = { 00, 00, 10, 05 }

Figure 3. Control flow graph and contents of code and data arrays of the virtualization-obfuscated example program, corresponding to the interpreter in Figure 1. Opcodes in the code array are printed in bold.

severe difficulties in producing *useful* results for the obfuscated program.

If the static analyzer is sound, it will always compute a result that is technically correct. That is, it will compute a valid over-approximation of the program semantics. At the very least, the static analyzer will compute the global invariant "true", i.e., the most imprecise element of the abstract domain. The challenge is to compute an over-approximation that is still precise enough to prove properties or provide useful information to a reverse engineer. In this work, we assume as the goal of static analysis a precise description of the set of arguments to an external function call that are possible in all executions of the program of interest.

3.1. Worked Example

We again look at how a static interval analysis processes the—now virtualization-obfuscated—program. The variables of the original program are no longer visible; but recall that we are interested in approximating the possible argument values at the site of the external function call. This requires the analysis to support array accesses. The length and initial contents of the arrays are known to the analysis, and we assume that the analysis precisely abstracts each element as an individual interval. The obfuscator is free to choose where to allocate local variables and constants in the data array and how to initialize them, but here we assume x and y were allocated at indices 0 and 1, respectively, and that the static data array is initialized to $\{00, 00, 10, 05\}$, as shown in Figure 3. We write d_x and d_y for `data[0]` and `data[1]`, respectively. As a parameter of `foo`, d_x can hold any value at the time of the function call. Finally, we compute the interval analysis as precisely as possible, i.e., we will not use an early widening step to accelerate finding a fixpoint but instead analyze several loop iterations precisely.

After the initialization of `vpc` to $[0; 0]$, the static analysis follows the program into the interpreter loop. The abstract state for line 6 of the interpreter is thus $\{vpc \mapsto [0; 0], d_x \in [-\infty; \infty], d_y \in [0; 0]\}$. The opcode of the first bytecode instruction stored at `code[0]` is `52`, with operands `01` (location of y) and `02` (location of the constant 10), corresponding to the initial assignment in line 2 of the original program (Figure 2). Therefore, the static analysis determines only the fourth case (assignment) of the switch statement to be reachable in the current state. Inside the case block in lines 26–31, the interpreter assigns the constant 10 to y using the local variables `op1` and `op2` to temporarily hold their data array indices. Finally, it increments the VPC by 3 to point to the next instruction. The static interval analysis is able to represent all these facts precisely. For each variable, it only determines a single value to be feasible in the current state. Upon leaving the case block to line 34, the state has become $\{vpc \mapsto [3;3], d_x \in [-\infty;\infty], d_y \in [10;10]\}$.

The next instruction is analyzed only in the next iteration of the interpreter loop. After computing the abstract successor along the back-edge of the `while`-loop from location 34 to 6, the analysis merges abstract states at line 6. The new abstract state at line 6 now becomes $\{vpc \mapsto [0;3], d_x \in [-\infty;\infty], d_y \in [0;10]\}$, over-approximating both the state before and after executing line 2 of the original program.

At this stage, the interval for the VPC already includes the operands of the previous bytecode instruction, whose values 01 and 02 can be misinterpreted as opcodes. A carefully designed analysis with a precise transfer function will still be able to deduce, however, that besides the fourth, only the first and default cases of the switch statement are feasible. By evaluating the assumption against the array contents, the analysis is able to narrow down `vpc` for each case. The default case exits the program, and the final state of the fourth case does not change. At the beginning of the first case, the abstract state at line 9 becomes $\{vpc \mapsto [3;3], d_x \in [-\infty;\infty], d_y \in [0;10]\}$. After evaluating all statements in the case block, the final state is $\{vpc \mapsto [5;5], d_x \in [0;0], d_y \in [1;11]\}$.

Again, this state is merged after processing the back-edge of the while loop, this time to yield $\{vpc \mapsto [0;5], d_x \in [-\infty;\infty], d_y \in [0;11]\}$ at line 6. Notice that the first case necessarily remains feasible. But now it can also be reached with a `vpc` of 5, so the edge `code[vpc]==03` can only restrict the value of `vpc` to $[3;5]$. Accordingly, `op1` is read from $[4;6]$ and determined to be $[1;3]$. The increment in line 10 can thus no longer be determined precisely, resulting in a *weak update*: the data array locations from 1 to 3 *may* have been incremented by 1. Besides y, this includes the constant used to initialize y and the relative offset used for the conditional jump. Therefore, the imprecision causes the analysis to lose track of which indices in the code array constitute proper opcodes. Furthermore, analysis also continues to loop through the first case, each time increasing the intervals for y and the constants. As a result, the analysis is unable to determine precise bounds for the call argument y.

3.2. Domain Flattening

The example shows that classic static analysis cannot achieve reasonably precise results on virtualization-obfuscated programs. Location-sensitive static analysis merges all domain elements at the same program location. In a virtualization-obfuscated program, all instructions of the original program, each of which was located at an individual program location, are interpreted using the same locations. Instructions of the same type share just the same case in the interpreter loop. But all instructions share the remainder of the interpretation loop. Therefore, *all* abstract states that would have been separate in the original program are merged into a single one in the obfuscated program. Therefore, *virtualization-obfuscation effectively reduces a location-sensitive static analysis to a location-insensitive one*. We call this effect *domain flattening*, as it removes one dimension of sensitivity from the abstract domain.

Thus it is clear that location-sensitive analysis, a reasonable trade-off between precision and cost in static analysis of regular programs, is unsuited for analyzing virtualization-obfuscated programs. In regular programs, the strategy of merging abstract states at locations is effective as long as all concrete program states at this location are sufficiently similar such that they can be summarized by a precise invariant. In virtualization-obfuscated code, invariants computed in the interpreter loop have to cover the entire execution of the original program. Regular abstract domains are not able to generate precise invariants for this scope.

The main insight that we are going to exploit in the next section is that virtualization obfuscation requires analyses not just to be location-sensitive, but also to be precise to a second dimension of program location—the virtual program counter.

4. VPC-Sensitive Static Analysis

We now show how to modify the components of an existing location-sensitive static analysis to work in presence of virtualization-obfuscation.

4.1. Lifting Abstract Domains

The main idea to make an analysis *VPC-sensitive* and hence robust against domain flattening is to equip its abstract domain $\mathbf{D}$ with an additional dimension of program locations for representing the virtual program counter. The value of the VPC is itself determined by the analysis and represented as an abstract value.

We lift $\mathbf{D}$ to VPC-sensitivity by mapping abstract VPC values to elements of $\mathbf{D}$. In the new domain $\hat{\mathbf{D}}$, each abstract state is then specific not only to the program location, but to the pair of location and VPC value. In the general case of a location-sensitive program analysis defined over an abstract domain

$\mathbf{D} = \mathbf{L} \to \mathbf{S}$, the lifted domain becomes $\hat{\mathbf{D}} = \mathbf{V} \to \mathbf{D}$, where $\mathbf{V}$ is the domain of abstract VPC values.

Many domains, such as the domain of intervals, maintain abstract states $\mathbf{S} = \mathbf{Var} \to \mathbf{U}$ as maps from variables to abstract values from some per-variable domain $\mathbf{U}$ and hence have the structure $\mathbf{E} = \mathbf{L} \to \mathbf{Var} \to \mathbf{U}$. For such domains, we can use the same domains for individual variables as for abstracting the VPC. We therefore set $\mathbf{V} = \mathbf{U}$ and define the lifted domain as $\hat{\mathbf{E}} = \mathbf{U} \to \mathbf{L} \to \mathbf{Var} \to \mathbf{U}$.

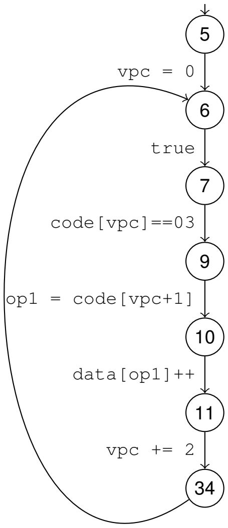
4.2. Extending the Transfer Function

The transfer function $\hat{f}$ for the lifted abstract domain is based on a pointwise extension of f to maps from abstract VPC values to elements of the original domain $\mathbf{D}$. In the same way as f_c is applied to each location separately in location-sensitive analysis, we here apply it separately to the abstract states mapped to pairs of locations and VPC values. It remains to define the computation of a successor VPC value from a triple $(\ell \mapsto v \mapsto s)$, so that the abstract successor state is mapped to the correct VPC value. In general, the transfer function can be freely defined as long as it over-approximates the concrete transfer function. But for the common case of abstract domains $\mathbf{E}$ mapping locations to maps from variables to abstract values, we now describe a method for automatically deriving the lifted transfer function for $\hat{\mathbf{E}}$.

Existing virtualization-obfuscators dedicate a particular register to storing the VPC (likely for performance reasons). This is not a fixed requirement, however; the VPC might be temporarily stored in various memory regions before it is used for reading the next instruction opcode. Therefore, we only require that the storage location of the VPC be unique for a particular program location. We further assume the existence of a function $vpcLoc(\ell)$ for reliably detecting the storage location of the VPC. If the VPC is not a fixed register, it could be detected by tracing the data flow from the location of switch jumps, following assignments during successor computation. In general, finding the VPC can also involve multiple refinement steps with candidate VPC locations or manual analysis.

We can now build the transfer function for a domain $\hat{\mathbf{D}} = \mathbf{U} \to \mathbf{L} \to \mathbf{S}$ with $\mathbf{S} = \mathbf{Var} \to \mathbf{U}$. Each domain element $v \mapsto \ell \mapsto s$ is computed by the transfer function from elements $v'_i \mapsto \ell'_i \mapsto s'_i$, such that

- $(\ell'_i, C, \ell) \in G$ is a control flow edge,
- $v = f_c(s'_i)(vpcLoc(\ell))$ is the new VPC value in the successor of s'_i, and
- $s = \bigsqcup_i f_c(s'_i)$ is joined from all abstract states with equal location ℓ and VPC v.



	Interval for y (data[1])			
	$vpc \in [3;3]$	$[5;5]$	$[10;10]$	$[12;12]$
5	$\bot$	$\bot$	$\bot$	$\bot$
6	$[10;10]$	$[11;11]$	$[12;12]$	$[12;13]$
7	$[10;10]$	$[11;11]$	$[12;12]$	$[12;13]$
9	$[10;10]$	$[11;11]$	$[12;12]$	$\bot$
10	$[10;10]$	$[11;11]$	$[12;12]$	$\bot$
11	$[11;11]$	$[12;12]$	$[13;13]$	$\bot$
34	$[11;11]$	$[12;12]$	$[13;13]$	$[12;13]$

Figure 4. Excerpt of the interpreter (case 1 for increment operations), as analyzed by a VPC-sensitive interval analysis. The table shows the two dimensions of abstract states, program location and abstract VPC values.

Here, $s(x)$ denotes the abstract value the variable x is mapped to in s. Note that the lifted transfer function applies the merging operator $\sqcup$ only for states at the same program location where the VPC evaluates to the same value, i.e., it is VPC-sensitive.

Example. Figure 4 shows the effect the VPC-lifting has on an interval analysis applied to the running example. For brevity, we just show case 1 of the interpreter loop, and only the abstract VPC values corresponding to the indices of the three increment instructions and the external call in the bytecode array. The special value $\bot$ marks unreachable combinations of program location and VPC: the VPC cannot be 3 before entering the loop at line 6; for a VPC of 12 (the location of the conditional jump opcode) the first case (lines 9–11) is not reachable, because control flows through the second case. At the end of the loop (line 34), the intervals are identical to those that the location-sensitive analysis determined for the respective locations in the unobfuscated program (see Figure 2).

4.3. Widening and Termination

We have to adapt an existing widening operator of the original analysis to ensure termination of the lifted analysis. An immediate way to define widening for the lifted domain is to apply the widening operator for the original domain in a pointwise manner. However, lifting creates a crossproduct of the original domain with the potentially infinite domain of VPC values, whereas program locations are always finite. This means that, while pointwise widening prevents infinite iterations for abstract states at identical VPC values and program locations, infinitely many different VPC values might accumulate at the same program location.

The VPC values encode the finitely many locations of instructions within the finite bytecode array. Intuitively, a finite number of abstract VPC values should therefore suffice to distinguish all indices. The VPC values are not guaranteed to point to valid array indices, however. An interpreter might compute the next valid VPC value using multiple steps, involving arbitrarily many intermediate values. Consequently, we need to allow widening across different VPC values to always guarantee termination of the analysis.

There are several options for implementing a suitable widening operator. It only has to satisfy the condition that it will reach a fixpoint within a finite number of steps. One possible strategy is to trigger widening once a certain threshold of VPC values has been exceeded; a reasonable bound can be given by the size of the static data present in the program. Widening can then proceed in two steps:

1) Once the threshold is exceeded for some location ℓ, all abstract states s_i with $v_i \mapsto \ell \mapsto s_i$ for all VPC values v_i are merged into a single state for a summary VPC, i.e., into $\bigsqcup_i^{\mathbf{V}} v_i \mapsto \ell \mapsto \bigsqcup_i s_i$, where $\sqcup^{\mathbf{V}}$ merges abstract VPC values and $\sqcup$ merges abstract states. Each new VPC value at ℓ is subsequently merged as well.
2) As new VPC values are merged into the summary VPC, regular widening is applied over the summary VPC to guarantee termination. In intervals, for instance, one bound can be set to infinity.

5. Lifting Bounded Address Tracking

We now give an adapted formalization of Bounded Address Tracking (BAT) [5], a static analysis targeted specifically to binaries, and apply our lifting to derive a VPC-sensitive variant.

5.1. BAT as Static Analysis

Abstract Addresses. The elementary abstract values in BAT are *abstract addresses*. Abstract addresses are pairs (r, o) consisting of an abstract memory region $r \in \mathbf{R}$ and an offset $o \in \mathbb{N}$. The set of memory regions $\mathbf{R}$ consists of the stack, allocated heap regions, and the global address space. Offsets are defined using integers for ease of exposition (the implementation supports bitvector values of different word lengths). Because of the lack of types in binaries, pointers are indistinguishable from integers. Therefore, regular integer values are represented as abstract addresses within the global address space. Memory region identifiers represent the statically unknown base address to which the respective region is allocated at runtime, which is 0 for the global address space.

For each region r, the special abstract address $(r, \top)$ represents an address where the offset is unknown within a region. This is used to abstract multiple offsets when a variable can take more values than are feasible to represent precisely. Finally, the single special value $(\top_R, \top)$ represents an unknown offset within an also unknown region. The set of abstract memory addresses $\mathbf{A}$ is thus defined as $\mathbf{A} = \{(\top_R, \top)\} \cup (\mathbf{R} \times (\mathbb{N} \cup \top))$.

Abstract Domain. Intuitively, BAT is similar to a concrete semantics. It is defined over a powerset of mappings from variables (registers and memory locations) to values (abstract addresses). These mappings are basically concrete machine states using abstract addresses instead of concrete addresses, i.e., if we assume an order over program variables, they are represented by vectors $\mathbf{A} \times \ldots \times \mathbf{A}$. An abstract state is a set of these vectors such that it over-approximates the reachable concrete states at a program location. Formally, the abstract domain is defined as $\mathbf{L} \to \mathcal{P}(\mathbf{A} \times \ldots \times \mathbf{A})$, i.e., a map from program locations to sets of vectors of abstract addresses. Merging of two states is defined as computing the union of the two vector sets.

Note that BAT differs from common abstract domains such as intervals: those domains map each location to a map from variables to abstract values, whereas BAT maps each location to a set of maps from variables to abstract values.

Transfer Function. The transfer function is defined as updating each individual value vector according to the effects of the respective statement. The updates follow the concrete semantics, with special rules for treating unknown offsets and regions (details can be found in [5]). The critical component to make the analysis feasible in practice is the value bounding that takes place when computing the successor states: for each variable, BAT only allows at most k distinct values across all value vectors in the abstract state. If a variable exceeds this bound, its value (r, o) is abstracted in two steps. First, all offsets are merged to $(r, \top)$. Second, if there are also more than k regions, all values are merged to $(\top_R, \top)$.

Effectively, BAT is thus relational, path and context sensitive up to a certain bound of variable values per location. This guarantees termination of the analysis by preventing an infinite accumulation of states in loops or recursive function calls. The variable bound can be adjusted according to the available computational resources. Path sensitivity prevents merging of information and thus allows a "brute-force" approach to the analysis of a virtualization-obfuscated program. Due to domain flattening, this approach requires high value bounds, however. Otherwise, BAT will have to merge values after a few iterations of the interpreter loop and will lose track of the values of code and data pointers.

5.2. VPC-Sensitive BAT

To equip BAT with VPC-sensitivity, the abstract domain and transfer function have to be adapted.

Abstract Domain. The lifted abstract domain for BAT is defined as $\mathbf{A} \to \mathbf{L} \to \mathcal{P}(\mathbf{A} \times \ldots \times \mathbf{A})$ and uses abstract addresses as the value domain for VPC values. To apply our lifting, we exploit that in regular BAT, an abstract state associated with a location is a set of vectors. The lifting to VPC-sensitivity is therefore equivalent to partitioning the set of vectors according to the value of the VPC in each vector.

Transfer Function. To implement the transfer function for individual abstract states, we modify the counting of abstract addresses per variable to respect the partitioning. VPC-sensitive BAT allows at most k distinct values across all value vectors in the same partition. If the bound is exceeded for a variable x at a VPC value v and location ℓ, all abstract address values for x in states s_i such that $v \mapsto \ell \mapsto s_i$ are merged. The new VPC value of each successor state is directly extracted from value vectors as explained in Section 4.2.

As discussed in Section 4.3, we need to modify the widening step in the lifted domain to guarantee termination. We introduce an additional bound m on the number of distinct VPC values tracked per location. Once this bound is reached, the VPC is widened in the same manner as regular abstract addresses. All states at this location whose VPC value is subsumed by the new summary VPC are joined and thus their value counts combined. This may lead to additional over-approximation of program variables. In practice, m should be set to a high (but finite) value to prevent premature widening.

6. Experiments

We now briefly discuss our prototype implementation and present preliminary experimental results on virtualization-obfuscated binaries.

6.1. Implementation

We implemented our analysis in the JAKSTAB static analysis platform for binaries [4]. JAKSTAB is based on the theory of abstract interpretation [2]. It over-approximates the semantics of a binary program taking the instruction fetch into account. Therefore, it does not require a control flow graph, which is not available for binaries. Instead, it processes the program beginning at the entry point of the executable, disassembling and analyzing the program one instruction at a time.

In each iteration of its analysis cycle, JAKSTAB reads the bytes pointed to by the current program counter value, decodes the next instruction, and translates it into an intermediate language. Branch instructions are then resolved into one or more control flow edges, based on the current abstract state. For indirect branches, the abstract state determines which target addresses are feasible. Non-branch instructions are simply transformed into a control flow edge pointing to the fall-through successor [6].

These explicit control flow edges allow JAKSTAB to compute successor states using the statement-wise transfer functions of a given abstract domain. The abstract states serve two purposes: they are used by the resolving-step to compute jump targets, and, as usual in static analysis, they are used to verify specifications. If the collection of all abstract states does not violate a property, the property is proven to hold.

JAKSTAB has been released as open source and can be obtained, along with a prototypical implementation of VPC-sensitive BAT, from http://www.jakstab.org.

6.2. Obfuscation and Analysis Targets

For our experiments, we relied on an obfuscation tool operating on source code that is part of an ongoing research project at the University of Arizona. Using a research obfuscator instead of VMProtect or CodeVirtualizer allows us to study virtualization-obfuscation in isolation without additional layers of protection and other noise. Our prototype does not provide a complete end-to-end solution for analyzing obfuscated binaries but instead focuses on the particular effects of virtualization. Nevertheless, we are working on extending our prototype to support in-the-wild obfuscated binaries.

As evaluation target we use a simple program for iteratively computing and finally printing the nth Fibonacci number, where n is supplied as a command line argument. The program is obfuscated and subsequently compiled to binary before analysis. To evaluate the precision of the analysis, we count the number of distinct possible arguments that the analysis can detect

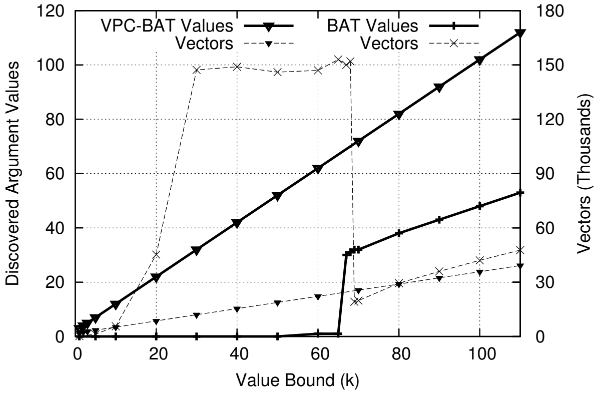


Figure 5. Discovered argument values and generated value vectors vs. value bound.

for the final external call to `printf`, before invariably widening the values (since the Fibonacci sequence grows to infinity, an exhaustive presentation of all values is impossible). This metric is representative for an application in malware detection, for example, where possible arguments to external system calls are evaluated against a malware specification [7]. In fact, most reverse engineering applications will have similar precision requirements, since at the very least they require a precise set of possible VPC values for each location. For instance, extracting a license check function requires to identify a precise sequence of VPC values that necessarily precede some event (such as creating a new window using an API call).

6.3. Results

We analyzed the obfuscated Fibonacci example with JAKSTAB using both standard BAT and VPC-sensitive BAT, with varying value bounds k. VPC-sensitive BAT encountered 108 distinct VPC values in all configurations. We used a VPC-bound of $m = 2000$ so the analysis never merged different VPC values.

Figure 5 shows the number of precise argument values discovered and the number of distinct variable vectors generated against k (each vector is a unique mapping of variables to abstract addresses). Table 1 provides additional detail for some of the data points. We can see that regular BAT is unable to yield any precise information about the call arguments below a value bound of $k = 60$, whereas VPC-sensitive BAT starts discovering values from $k = 1$ and shows a linear relation between k and the number of argument values. In fact, the number of different arguments detected is $k + 2$; this is equal to the two base cases, which are checked explicitly, and one additional value per tracked value in the iteration. Regular BAT also starts discovering arguments from $k = 60$ on. From $k = 68$ also linearly, but at a slower rate of about $\frac{k}{2}$ and at a higher cost in the number of value vectors generated. Between bounds 30 and 68, regular BAT times out after 5 minutes, having generated about 150'000 vectors. For these bounds, the information loss from widening leads to exploring many infeasible paths through the interpreter. Above 68, regular BAT prevents the information loss by the "brute force" of fully path-sensitive analysis.

Table 1. Detailed results

Analysis	k	Args	Vectors	Time	Mem (MB)	Sound
BAT	10	0	5610	4s	68.7	n
BAT	40	0	149000	>5m	129.3	n
BAT	60	1	147000	>5m	134.7	n
BAT	70	32	19800	13s	75.9	n
V-BAT	1	3	2121	3s	19.7	y
V-BAT	10	12	5181	4s	27.2	y
V-BAT	30	32	11981	10s	107.5	y
V-BAT	70	72	25581	40s	71.2	y

In all experiments, however, regular BAT was unable to resolve several memory writes and had to perform weak updates to the global address space. Under strict semantics, this weak update also affects the static data area of the program, including imports; the analysis was unsound in that it could not explore the control flow to an address that was potentially overwritten. The analysis thus failed to compute a sound over-approximation of the program behavior. VPC-sensitive BAT, on the other hand, finished with a sound and precise over-approximation in all cases.

As with all Java programs, JAKSTAB's memory usage statistics reported in Table 1 should be taken with a grain of salt. Due to garbage collection, there is no linear relationship between the number of vectors and the allocated heap memory. We can see, however, that VPC-sensitive BAT incurs a memory cost comparable to, or even lower than, regular BAT.

7. Related Work

Rolles [9] discussed how to deobfuscate virtualized programs, showing that the protection offered by virtualization-obfuscations is vulnerable to manual attacks. Sharif et al. [10] presented a pioneering approach to automatic reverse engineering of such programs. They determine the semantics of bytecode instructions in order to deobfuscate the program. These kinds of approaches face difficulties with obfuscators that generate bytecode instructions at different granularities. In general, deobfuscation of virtualized bytecode with the purpose of deriving a regular binary is similar to *decompilation*.

Coogan et al. [1] showed how to successfully apply dynamic analysis to detect code influencing system call arguments in virtualization-obfuscated binaries. They analyze dependencies within a single trace, with the goal of distinguishing the dependencies within the interpreter from those due to the original program. Their work is unique in that it abstracts from the specific virtualization engine used. Their detection of calls and returns contains promising ideas for adapting interprocedural analyses, which we plan to investigate in future work. However, the approach suffers from the same incompleteness as any dynamic analysis; code that was not executed is not subsequently analyzed.

The net effect of virtualization-obfuscation is similar to *control-flow flattening*. Control-flow flattening splits the program into basic blocks and links them via a dispatcher loop and a virtual program counter (called *dispatcher variable* [11]). The main difference to virtualization is that program locations are not folded, i.e., for each original program location, there is one in the obfuscated program. Thus, only the dispatcher loop can cause merging of unrelated abstract states. Udupa et al. [11] showed that this can be resolved by cloning the shared program locations and using constant propagation to trace the dispatcher variable. In virtualization, however, the additional sharing of code for individual instructions prevents this approach from being effective. Due to loops, a potentially infinite number of locations would be cloned if one were to apply the same approach directly.

Finally, the idea of lifting an abstract domain to increase its precision is also found in the trace partitioning domain [8], which adds a degree of path sensitivity to an existing analysis. ESP [3] similarly improves the precision of a static analysis by lifting it with respect to the states of a property automaton.

8. Conclusion

Virtualization-obfuscators pose a significant challenge, but their specific effects can be precisely characterized. We showed how to add an additional dimension to the location-sensitivity of an existing static analysis to make it robust against this kind of obfuscation. Experiments on a toy example show promising results; we are working on extending the applicability of our prototype to in-the-wild code examples.

Acknowledgments. The author would like to thank Christian Collberg and Sam Martin for generously providing the obfuscated code sample.

References

[1] K. Coogan, G. Lu, and S. K. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proc. ACM Conf. Computer and Communications Security (CCS 2011)*, pages 275–284. ACM, 2011.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. 4th ACM Symp. Principles of Programming Languages (POPL 1977)*, pages 238–252, Jan. 1977.

[3] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. 2002 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 2002)*, pages 57–68. ACM, 2002.

[4] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Proc. 20th Int. Conf. Computer Aided Verification (CAV 2008)*, volume 5123 of *LNCS*, pages 423–427. Springer, 2008.

[5] J. Kinder and H. Veith. Precise static analysis of untrusted driver binaries. In *Proc. 10th Int. Conf. Formal Methods in Computer-Aided Design (FMCAD 2010)*, pages 43–50, 2010.

[6] J. Kinder, H. Veith, and F. Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proc. 10th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[7] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive detection of computer worms using model checking. *IEEE Trans. Dependable Sec. Comput.*, 7(4):424–438, Oct. 2010.

[8] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *14th European Symp. Programming (ESOP 2005)*, volume 3444 of *LNCS*, 2005.

[9] R. Rolles. Unpacking virtualization obfuscators. In *Proc. Workshop On Offensive Technologies (WOOT 2009)*. USENIX, 2009.

[10] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 94–109. IEEE Computer Society, 2009.

[11] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conf. Reverse Engineering (WCRE 2005)*, pages 45–54. IEEE Computer Society, 2005.