Data structure archaeology: scrape away the dirt and glue back the pieces!

(Or: automated techniques to recover split and merged variables)

Asia Slowinska, Istvan Haller, Andrei Bacs, Silviu Baranga, and Herbert Bos

Vrije Universiteit Amsterdam

Abstract

Many software vendors use data obfuscation to make it hard for reverse engineers to recover the layout, value and meaning of the variables in a program. The research question in this paper is whether the state-of-the-art data obfuscations techniques are good enough. For this purpose, we evaluate two of the most popular data obfuscation methods: (1) splitting a single variable over multiple memory location, (2) splitting and merging two variables over multiple memory locations. While completely automated and flawless recovery of obfuscated variables is not yet possible, the outcome of our research is that the obfuscations are very vulnerable to reversing by means of automated analysis. We were able to deobfuscate the obfuscated variables in real world programs with false positive rates below 5%, and false negative rates typically below 10%.

1 Introduction

Both malware authors and commercial software vendors employ software obfuscation to protect their binaries from the prying eyes of reverse engineers and crackers. The assumption is that sensitive information is safe behind one or more layers of transformation that scramble the data and code in such a way that they become hard to analyze. In this paper, we focus on legitimate applications written in C that vendors obfuscate for purposes like IP protection and DRM.

By now, code and data obfuscation have evolved into mature fields, with an active research community and commercial products like Irdeto's Cloakware [19], Morpher [26], and CodeMorph [31]. The commercial interest in obfuscation is high, especially in DRM or security-sensitive environments. Cloakware's clients include companies like Logitech/Google TV, ComCast, Netflix, Elgato, Harmonix, and Xceedium, while Morpher has clients like Spotify and Discretix (used in Android DRM, Microsoft PlayReady, and many other products).

The obfuscation techniques in today's obfuscators range from limited control flow hiding to highly advanced methods that include the data structure layouts (and values) as well. Indeed, it is common to distinguish between control obfuscation (e.g., opaque predicates and control flow hiding), and data and layout obfuscation (e.g., splitting a single variable over multiple memory locations) [10, 9].

Over the years, both the blackhat and whitehat communities have shown an active interest in probing the strength of current control obfuscation techniques. Typically, they show that most control obfuscation techniques are limited (or even weak) in the face of determined attackers [21, 33, 24, 13].

We do not know of any project that addresses the recovery of obfuscated memory layouts and data. This is remarkable, because for reverse engineers there is great value in the recovery of the data and its layout. Real programs tend to revolve around their data structures, and ignorance of these structures makes the already complex task of reverse engineering even more painful [30]. In addition, deobfuscated data is a crucial step in reversing sensitive information.

A possible reason for the lack of prior art is that extracting data structures from binary programs is exceedingly difficult even without obfuscation. If data is additionally hidden behind sophisticated obfuscations, the hope of recovering the original data structures is close to zero. For instance, how could you detect that two bytes are really part of the same number in a single structure, if they are not even *stored* together? The core assumption that many software vendors rely on is that the obfuscation is *irreversible in practice*. The research question in this paper is whether this assumption is reasonable. Specifically, we show that the assumption is false for state-of-the-art data obfuscators and that it is feasible to recover the data structures in an automated way. Perfect deobfuscation is not needed and, as we shall see, impossible in the general case. Instead, we probe the binary's obfuscation and consider it weak when a reverse engineer has a high probability of finding the original data/layout.

Contributions. The research question in this paper is: are state-of-the-art data obfuscation techniques good enough? Specifically, we probe two of the most common and advanced data obfuscation techniques, and show that they are vulnerable to automated analysis. Our approach is based solely on a dynamic analysis of the program's memory accesses and information flow. As a concrete implementation, we present Carter, a data deobfuscator that reverses the obfuscations by tracking and analyzing the program's memory access patterns. We also show the usefulness of the tool in an actual reverse engineering scenario.

Assumptions. We assume that the obfuscator may apply different data and control obfuscations at the same time—much like state-of-the-art obfuscators such as Cloakware [19]. For instance, we allow split variables in a program that also runs on a virtualization obfuscator with opaque predicates. Rather than criticize a specific product, our aim is to evaluate the advanced obfuscation techniques in general, regardless of who sells them. Therefore, whenever we discuss data obfuscation techniques in this paper, we refer to publications describing the techniques and not to products. The actual obfuscator used for this paper is representative of the advanced data obfuscations found in (one or more of) the commercial systems.

We also assume that the obfuscated binary is available on a machine controlled by the attackers. They can apply any kind of static or dynamic analysis to the binary and run it many times.

Our approach is almost exclusively dynamic. In particular, it runs the obfuscated binaries in the PIN [18] dynamic instrumentation framework. Dynamic analysis has the advantage that it easily handles popular control obfuscations (like opaque predicates and return address patching, see Section 5). The drawback is that, like all dynamic approaches, we can only analyze what we execute. While code coverage for binaries is a hot research topic [6], it is beyond the scope of this paper.

The paper will also show-case examples of single-threaded obfuscation, since in practice, we never observed instances cross-thread data obfuscation. This does not limit the ability of Carter to handle multi-threaded programs, as PIN is fully capable of performing per-thread instrumentation.

Outline We describe data obfuscation techniques in Section 2, and our approach to deobfuscation in Sections 3-4. Next, we discuss the impact of control obfuscations in Section 5 and evaluate our work in Section 6. We then use our tool in an actual reverse engineering example in Section 7 to demonstrate its usefulness. We discuss both limitations and recommendations in Section 8 and related work in Section 9. Section 10 contains our conclusions.

2 Data obfuscation

In the next three sections, we focus on data obfuscation. In Section 5, we also show what happens if data and control obfuscation are combined. Our focus is solely on *obfuscation* rather than, say, encryption[1]. Specifically, we evaluate two of the most prevalent and advanced data obfuscations described in the literature:

- *Variable splitting*: the program scatters variables (like integers) over multiple locations.
- *Splitting and merging*: besides splitting variables, the program merges them by using a single location for multiple variables. While we are not aware of any current obfuscator that provides a flexible manner to add such data obfuscation, and we were able to perform only a partial evaluation, we think it represents an interesting extreme case.

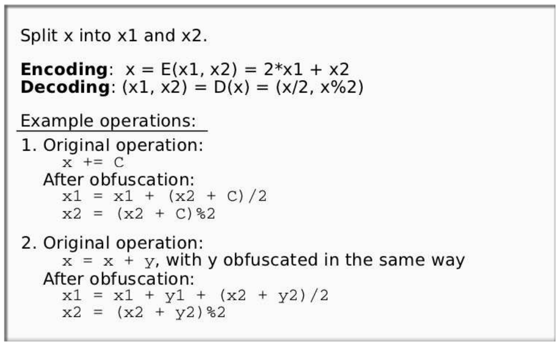
We now discuss these techniques in detail and focus on transformations that obscure the built-in data types [11].

Splitting variables Integers and boolean variables are common data types that often carry sensitive information. A popular and complex transformation is known as *variable splitting*. Splitting a variable breaks it up into several smaller components. Wherever possible, the program will access the constituent components rather than the actual parameter. For reverse engineers it will be very difficult to guess the meaning of the components.

To introduce the concept formally, assume that the obfuscator splits a variable `x` into variables x_1 and x_2, with the transformation defined by functions $E(x_1,x_2)$ and $D(x)$. We say that `E` *encodes* `x` as a function of x_1 and x_2, while `D` *decodes* `x` (maps `x` on the corresponding values of x_1 and x_2). Figure 1 shows an example.

Given `E` and `D`, we still have to devise operations to perform on the new representation of `x`. A simple solution would be to compute the value of `x`, perform the original

[1] Variable *encryption* is not normally seen as obfuscation as it involves secret keys rather than key-less obfuscation algorithms. Thus, it is the subject of cryptanalysis rather than deobfuscation.



Split x into x1 and x2.

Encoding: x = E(x1, x2) = 2*x1 + x2
Decoding: (x1, x2) = D(x) = (x/2, x%2)

Example operations:

```
1. Original operation:
     x += C
   After obfuscation:
     x1 = x1 + (x2 + C)/2
     x2 = (x2 + C)%2

2. Original operation:
     x = x + y, with y obfuscated in the same way
   After obfuscation:
     x1 = x1 + y1 + (x2 + y2)/2
     x2 = (x2 + y2)%2
```

Fig. 1. An example variable split transformation.

operation on x, and encode it as x_1 and x_2 again. However, doing so reveals the variable x to the attacker. Split transformations try to perform the operations solely on the new representation of the variable and avoid computing x even as an intermediate value. Figure 1 shows an example that maps additions on operations on x_1 and x_2.

Of course, we cannot always hide x completely. When the program passes x as an argument to a library function or a system call, it needs to compute its value. In general, all interactions with non-obfuscated code require x's original value. Also, while the potency of the split obfuscation grows with the number of new variables introduced, so does the cost of the transformation. In practice, a variable is split into just 2 or 3 other variables [11].

Splitting and merging variables To add to the confusion, the obfuscator may combine splitting and merging on the same variables: given unrelated variables x and y, the transformation first splits x into $\{x_1, x_2\}$, and y into $\{y_1, y_2\}$. Next, it merges x_2 with y_1 into z, so the obfuscated program uses only variables x_1, z, and y_2.

2.1 Goal: tractable deobfuscation

Carter aims to make data deobfuscation tractable—to give reverse engineers a high probability of finding the original data. Without knowing the original intention of the programmer, it is not always possible to decide whether a variable is obfuscated, or encoded in a certain way for other reasons. For example, to access a two-dimensional array, `arr`, a programmer may use either one or two subscripts, i.e., `arr[x][y]` or `arr[i]` where $i = x{*}N{+}y$. Thus, when we observe such array accesses in a binary, we cannot tell whether the programmer chose the encoding for convenience, or to obfuscate the variable i by splitting it into variables x and y.

In our analysis, we just aim to discover that x and y are used interchangeably with i. After the analysis, we will then explicitly compare our results with the ground truth and report the variables that were not really obfuscated as false positives (Section 6).

We stress that perfect deobfuscation is not needed. Specifically, we can tolerate false positives (where we say that data was obfuscated, when in reality it was not) and even false negatives (where we miss the obfuscation), as long as these cases do not occur too often. The reason is that the number of variables in a program may be large, but it is only a fraction of the total SLOC count. For instance, the `lighttpd` webserver used by YouTube counts about 2k variable/field definitions on 40K lines of code. Even if we incur a false positive rate of 5%, the number of false positives for programs like `lighttpd` is probably tractable for a motivated attacker. Phrased differently, the base-rate fallacy [3] is less of an issue than for, say, most intrusion detection systems. Similarly, a false negative rate of 10% means that we miss obfuscated variables, but the remaining 90% are important results. Thus, the real question is whether a reverse engineer can use automated techniques to get a handle on the data and layout.

3 Variable split detection

To detect a split variable, we build on two observations. First, when an obfuscator splits a variable `z` into `x` and `y`, it needs to perform a semantically equivalent operation on `x` and `y` for all operations on `z` (whether they be reads, writes, or ALU operations). Second, although the obfuscator works on `x` and `y` independently as much as possible, their values are combined occasionally. For instance, during an interaction with non-obfuscated components, such as the operating system.

Carter therefore analyzes the program's memory access trace, and looks for variables that are used together and exchange their data locally—in a short logical time interval. The question is how to determine the right level of affinity. For this we developed a new approach that hails from a technique in cache optimization.

Reference affinity grouping [37] restructures arrays and data structures to place elements that are always used together on the same cache line. It measures how 'close in logical time' the program accesses groups of data, and proposes a partition based on the outcome. Likewise, Carter looks for candidate data items that together may make up a split variable by tracking items that are used close together in logical time. Whenever Carter finds such items, it classifies them for a grouping.

Although we were inspired by the original work on reference affinity grouping [37], we devised our own method for approximating the solution. Picking an appropriate method is important, because Ding et al. [15] proved that finding the optimal partition is NP-hard. The concept of *temporal reuse intervals*, which we propose in Section 3.3, provides a practical way to identify memory locations that are accessed together.

Once the grouping algorithm has proposed candidates for split variables, we refine the results by data flow analysis (Section 3.4). Intuitively, data items in a split variable share data on reads.

Running example We illustrate the whole procedure with a simple obfuscated function, which serves as a running example. For the sake of clarity, all examples are in `C`. However, we perform the real analysis on binaries.

The code in Fig. 2.a computes the factorial of the input variable `n`. We apply the transformation in Fig. 1 to split the loop variable `i` into `j` and `k` (Fig. 2.b). The obfuscation is admittedly simple (and thus easy to understand), but the analysis works exactly the same for more complex connections between `j` and `k`. After all, regardless of the exact obfuscation, the code would still use `j` and `k` "together" and they would exchange information to interact with non obfuscated parts of the environment.

```
    int factorial (int n) {
[1]  int f = 1, i;
[2]  for (i = 0; i < n; i++)
[3]    f = f * (i + 1);
[4]  printf("factorial(%d)=%d\n", n, f);
[5]  return f;
    }
```

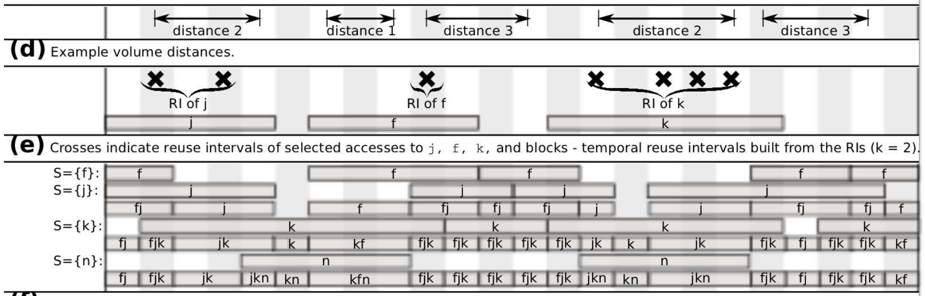
(a) The function before obfuscation.

```
    int factorial (int n) {                    Access instructions:
[1]  int f = 1;                                f: 0,
[2]  int j = 0, k = 0;                         j: 1. k:2,
[3]  while (2*j+k<n){                          j:3, k:4, n:5,
[4]    k = (k+1)%2;                            k:6, k: 7,
[5]    if (!k) j++;                            k:8, j:9,
[6]    f = f*(2*j+k);                          f:10, j:11, k:12, f:13,
[7]  }
[8]  printf("factorial(%d)=%d\n", n, f);       n:14, f:15,
[9]  return f;                                 f:16
    }
```

(b) The function after obfuscation. The numbers on the right count access instructions. E.g., in line [3], we have accesses to: j: 3, k: 4, n: 5.

Time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Instruction:	0	1	2	3	4	5	6	7	8	10	11	12	13	3	4	5	6	7	8	9	10	11	12	13
Variable:	f	j	k	j	k	n	k	k	k	f	j	k	f	j	k	n	k	k	k	j	f	j	k	f

(c) An access trace generated by `factorial()`.


distance 2
distance 1
distance 3
distance 2
distance 3
(d) Example volume distances.
RI of j
RI of f
RI of k
j
f
k
(e) Crosses indicate reuse intervals of selected accesses to j, f, k, and blocks - temporal reuse intervals built from the RIs (k = 2).
S={f}:
S={j}:
S={k}:
S={n}:
fj
fjk
jk
k
kf
jkn
kn
kfn

(f) The procedure of merging TRIs build from RIs (k=2). Each TRI block indicates which variables form the set S associated with it.

({f, j, k}, {{0, 1, 3, 2, 4, 6, 7, 8}, {10, 11, 2, 4, 6, 7, 8}, {10, 11, 12}, {13, 11, 12}, {13, 3, 12}, {13, 3, 2, 4, 6, 7, 8}, {10, 9, 11, 2, 4, 6, 7, 8}, {10, 9, 11, 12}, {13, 9, 11, 12}})
({j, k, n}, {{1,3, 2, 4, 6, 7, 8, 5}, {3, 2, 4, 6, 7, 8, 5}, {9, 11, 2, 4, 6, 7, 8, 5}})
({k, f, n}, {{2, 4, 6, 7, 8, 10, 5}})

(g) The resulting TRIs - after dropping the TRIs included in other TRIs. Carter does not keep track of the time intervals anymore.

(h) The final step is to drop variables which are at times accessed individually, far from other group members. Since later `factorial()` accesses f separately (the `printf` function), we discard sets {f, j, k}, {f,j}, {f, k}, and {j, k} is the result of the analysis.

Fig. 2. An example of the variable split detection procedure.

3.1 Usage patterns

Carter's detection procedure revolves around *usage patterns*—sets of memory locations accessed together. Consider the following memory accesses by factorial(): `fjkjknkkkfjkfjkn...` (Fig. 2.c). Depending on the input, this sequence might grow arbitrary large, and contain a lot of redundant information. Once the loop has started, we expect cycles of accesses `jknkkkfjkf`, possibly including an access to `j`. A useful and compact representation would indicate {`j`,`k`} or {`j`,`k`,`f`} as common *usage patterns*. Likewise, we can make a pattern with `n`, which is less common.

In the beginning of this section, we observed that the components of a split variable are accessed close to each other. To find split variables, we therefore look for usage patterns. Carter treats these patterns as crude candidates for a split transformation.

3.2 Reference affinity grouping

We now formalize the concept of usage patterns by means of the *reference affinity model* [37].

An *access trace* T is a sequence of memory accesses over time; we assign a logical time to each of its elements. For instance, `factorial` in Fig. (2.c) may access the following sequence of variables `fjkjknkkkfjkfjkn...` in its access trace. We use $\mathsf{a}_{\mathtt{f}}$ to denote an access to `f`, and trace element $\mathrm{T}[\mathsf{a}_{\mathtt{f}}]$ represents the logical time of $\mathsf{a}_{\mathtt{f}}$.

Given two accesses $\mathsf{a}_{\mathtt{x}}$ and $\mathsf{a}_{\mathtt{y}}$ in a trace T, we define the *volume distance* as the number of distinct data elements accessed in times T[$\mathsf{a}_{\mathtt{x}}$], T[$\mathsf{a}_{\mathtt{x}}$]+1,. . ., T[$\mathsf{a}_{\mathtt{y}}$]-1, and we write dist($\mathsf{a}_{\mathtt{x}}$,$\mathsf{a}_{\mathtt{y}}$). Observe that the volume distance differs from the time distance. For example, the volume distance between the accesses to `x` and `y` in the trace `xfooy` is 3 (elements: `x`, `f`, `o`), dist($\mathsf{a}_{\mathtt{x}}$,$\mathsf{a}_{\mathtt{y}}$) = 3, while the time distance is 4.

Definition 1. *We define a linked path with link length k as a sequence of accesses to distinct data elements where the volume distance between each two consecutive accesses is less than k.*

Later, we will restrict the elements of a linked path to be members of a set S, as in the following definition of strict reference affinity.

Definition 2. *Given an access trace T, a set S of memory locations is a strict reference affinity group with link length k if and only if (1) for each location $\mathtt{x} \in$ S, all its accesses $\mathsf{a}_{\mathtt{x}}$ have a linked path from $\mathsf{a}_{\mathtt{x}}$ to an access to $\mathtt{y}$, for any $\mathtt{y} \in$ S, and (2) the set S is maximum, i.e., it cannot be extended without invalidating condition (1).*

Intuitively, all members of an affinity group are always accessed together, they are "close" to each other – for each memory access, we can find a path linking it to an access to any other element in the set. As shown in [37], for a given access trace and a link length k, the affinity groups form a unique partition of the program data.

Since the memory locations that are the result of a split operation are always used together, we expect that even for small values of k, we will consider them as a part of the same group. Ideally, we would like to obtain the k-affinity groups. However, as accurately computing the reference affinity groups is an NP-hard problem [15], in Sections 3.3-3.4 we propose a new heuristic to find them.

3.3 Temporal reuse intervals (TRIs)

In this section, we introduce temporal reuse intervals. We additionally illustrate the novel concepts with the access trace of Fig. 2. Later, in Section 3.4, we use temporal reuse intervals to approximate the reference affinity groups for a memory access trace.

Given an access trace T, and a link length k, consider accesses to memory locations `x` and `y`. An access $\mathsf{a}_{\mathtt{x}}$ is a *remote usage* if it is "far" from the previous access to `x`. More formally, if we denote the previous access to `x` by $\bar{\mathsf{a}}_{\mathtt{x}}$, then dist($\bar{\mathsf{a}}_{\mathtt{x}}$, $\mathsf{a}_{\mathtt{x}}$) $> k$.

Definition 3. *A reuse interval RI of `x` is a maximal sequence of accesses to `x` where (only) the first is a remote usage.*

Intuitively, a reuse interval is a set of "all" accesses to `x` that are close to each other.

Since reuse intervals relate to single memory locations, we combine them so that we can reason about multiple locations at the same time. This combination is known as *a temporal reuse interval* (TRI). We first define it, and next we show the connection between reuse intervals and temporal reuse intervals: we construct a TRI from an RI, and we explain how to merge two TRIs.

Definition 4. *A temporal reuse interval TRI = (I, S, P) is a tuple of a time interval I, a set S of memory locations, and a set P of instruction addresses such that*

- *for each access a_z in the time interval I (i.e., T[a_z] $\in$ I), where z may or may not be $\in S$, and*
- *for each memory location x in S (i.e., $x \in S$),*

there exists a_x, an access to x, realized by an instruction $p \in P$, such that the accesses are close to each other, i.e., dist(a_z,a_x)$\leq k$.

In other words, a TRI guarantees that all accesses in the time interval I are close to all memory locations in S.

Given a reuse interval RI of x, we can construct a TRI. Let I be the time interval associated with RI, and P the set of instructions that access x in RI. We extend I backward and forward as long as all accesses in the new interval $\bar{I}$ are close to an access to x. That is, for each access a_z, T[a_z] $\in \bar{I}$, there is an access to x such that dist(a_z,a_x) $\leq k$. Finally, ($\bar{I}$, $\{x\}$, P) is the new TRI. Refer to Fig. 2.e for an example.

We now explain how we merge two TRIs which overlap in time. Given (I_0, S_0, P_0) and (I_1, S_1, P_1), we combine them to obtain the following new ones: ($I_0 \setminus I_1$, S_0, P_0), ($I_0 \cap I_1$, $S_0 \cup S_1$, $P_0 \cup P_1$), and ($I_1 \setminus I_0$, S_1, P_1). We discard the empty sets. For an intuitive explanation refer to Fig. 2.f.

3.4 From TRIs to split variable detection

To propose candidates for split variables, Carter computes temporal reuse intervals. Next, it refines the results by selecting the candidates that are always accessed together, and not only in some of the instructions. Finally, we confirm that a fitting dataflow exists between the candidates, and we output the resulting split variables.

Generating candidate sets Carter classifies memory locations according to their allocation time, and calculates temporal reuse intervals for each of these groups individually. It assigns a unique allocation time to each function frame on the stack, each object allocated on the heap, and the data segment of a binary. We never consider memory locations with different allocation times for a TRI grouping.

To construct TRIs, Carter first calculates reuse distances and remote usages for all memory locations, so that it can determine reuse intervals. Incidentally, since precise reuse distance computation would be expensive in terms of memory, we implemented the approximation proposed by Ding et al [16], which yields very good results while requiring only logarithmic space.

Having determined the reuse intervals, Carter extends them backward and forward, and constructs temporal reuse intervals. Next, it merges TRIs that are not disjoint and drops TRIs for which the memory locations and instruction addresses are already included in other TRIs[2]. Fig. 2.f-2.g illustrate the procedure.

Refining the candidate sets The previous step computes sets of memory locations that the program accessed within a bounded (volume) distance. Carter refines the sets by discarding these ones whose elements are at times accessed individually, far from other

[2] At this point, we care about candidates, not time intervals anymore

group members. It makes sense as we expect the program to access the components of a split variable together.

Dataflow confirmation Memory locations that originate from a single variable share data on interactions with the unobfuscated components of the system. For example, the binary combines their values before they turn into an argument to a system call or a library function. Thus, generating the candidates for a split, Carter confirms that a flow of data exists between them.

When the obfuscator decodes the original variable, it combines values of the split components. To detect this transformation, Carter assigns colors to the split candidates determined in Section 3.4, and employs dynamic taint analysis [20] to check if the colors are combined.

Fig. 2.h presents the variables classified by Carter as split. Observe that in this case, `j` and `k` are not combined during an interaction with a non-obfuscated component, but during the comparison with `n`.

4 Combined split and merge

In theory, we can make variable splitting more powerful by also *merging* variables. Given unrelated variables `x` and `y`, the transformation first splits `x` into $\{\mathtt{x}_1, \mathtt{x}_2\}$, and `y` into $\{\mathtt{y}_1, \mathtt{y}_2\}$. Next, it merges $\mathtt{x}_2$ with $\mathtt{y}_1$ into `z`, so the obfuscated program uses only variables $\mathtt{x}_1$, `z`, and $\mathtt{y}_2$. In other words, `x` and `y` 'share' a component variable `z`.

Even though we are not aware of any current obfuscator that provides a flexible manner to implement such (complex) data obfuscations, we added a detection module for it and verified that it works on a limited set of examples. However, as we were not able to combine this obfuscation with Control obfuscations (refer to Section 5), we did not evaluate the strength of the split+merge obfuscation extensively.

To detect the combined split+merge obfuscation, we use a technique that is similar to that of split detection. The main departure is that it looks for different usage patterns, but all steps up to and including itemset selection are the same. However, rather than simply eliminating all patterns that contain elements that do not always appear together, the split+merge detection module uses selection criterion that is slightly different.

We say that $x \prec y$ if y is also accessed when x is accessed. If $x \prec y$ and $y \prec x$ we say that $x = y$. A pattern xyz is valid if $x \preceq z$ and $y \preceq z$. In split+merge, Carter eliminates all patterns that cannot be written in this way. After this, we keep (only) the maximal patterns that reach this point. So if S1 is a subset of S2, we eliminate S2. The final step is again the dataflow confirmation, which is exactly the same as for the split obfuscation.

5 Adding control obfuscation

Obfuscators often combine data obfuscation with control obfuscation such as opaque predicates, return address patching, and virtualization with instruction set modification. We do not target control obfuscations at all, but we briefly discuss the influence of some popular techniques that an obfuscator may apply *in combination* with the above data obfuscations.

5.1 Control obfuscation

In addition to data obfuscation, obfuscators often apply one or more of the following control obfuscation techniques:

- *Opaque predicates* are code sequences that are hard/impossible to analyze statically, but always produce the same results at runtime. The static analyzer is obliged to consider a huge number of possible outcomes. As a result, the analysis becomes inaccurate and often intractable. For instance, the program may calculate a jump target using an opaque predicate. If it cannot determine the outcome, a static analyzer has to consider all possible addresses as jump targets.
- *Return address patching* is a technique whereby functions dynamically change their return addresses, so that they return not to the instructions following the call, but a few bytes further. The modified control flow confuses advanced disassemblers like IDA Pro.
- *Control flow flattening* transforms the program's well-structured control flow graph. Thus is typically done by replacing all call instructions by indirect jumps and adding a single dispatcher that maintains all control flow.
- *Virtualization* means that the program consists of bytecode that is interpreted by a tailored VM. Thus, the code in the binary file has no correspondence to the program code itself. Moreover, the bytecode's instruction set may be different from that of the host. Well-known commercial virtualization obfuscators include VMProtect and Code Virtualizer [1].

5.2 Preventive transformation

Preventive transformation are not obfuscations *per se*, but they make it harder to recover the original data. Besides the proper obfuscations, we augment the obfuscator with a preventive transformation that is specifically tailored to derail Carter.

- *Memory access injection* adds instructions that introduce spurious data accesses and calculations. As Carter relies on memory access pattern analysis, such accesses make our analysis more difficult and less precise.

5.3 Impact of control obfuscation

In this section, we discuss to what extent the preventive transformation and control obfuscation hinder Carter.

Since Carter builds on dynamic analysis rather than static analysis, Carter does not really suffer from the first two control obfuscations at all. At runtime, we encounter solely the actual outcomes of opaque predicates and return addresses—there can be no confusion. The only effect that may occur is that the opaque predicates introduce new memory accesses that modify the memory access patterns that serve as inputs for Carter's analysis, specifically for the detection of split variables.

Control flow flattening also has little effect on our analysis as Carter has no interest in the control flow graphs itself. Instead, it considers only the program's memory accesses to read or write data. Again, there may be a small effect if the flattened control flow introduces new memory accesses.

Virtualization makes it harder to analyze the instructions and their meanings. However, previous work has shown how to identify instructions that are part of the original code [13]. This is good enough for Carter. Our analysis relies solely on the program's *memory access patterns*. As long as we can identify accesses to data that are due to the program's instructions (rather than the interpreter), our method still works.

Of course, the interpreter may well generate additional data accesses that we cannot easily filter out. Again, such 'spurious' memory accesses may confuse our analysis. Phrased differently, virtualization itself is not really a problem for our analysis, but the spurious memory accesses might be.

We conclude that in all cases, the modified and added memory access patterns do influence Carter's detection of split, or split and merged variables, but the control flow itself is not important. Memory access injection is a program transformation that encapsulates exactly this effect. It is specifically tailored to derailing Carter's analysis. In Section 6 we evaluate the effect of spurious memory accesses (introduced by whatever obfuscation or transformation) on our analysis.

6 Evaluation

To evaluate our approach, we apply it to a set of eight stripped and obfuscated Linux applications. Since we use dynamic analysis, we can classify only the memory that the program accesses during the experiments. We use the applications' normal test suites as inputs and combine the results of multiple runs of the binaries to increase the coverage of both the code and the data. Our experiments include four real world applications (*lighttpd* [40K LoC[3]], *wget* [36K LoC], *grep* [21K LoC], and *gzip* [19K LoC]), and four CoreUtils (*ls*, *base64*, *expr* and *factor*).

To determine whether or not Carter helps reverse engineers to recover obfuscated data structures, we focus our evaluation on the number of variables Carter recovers, as well as the number of false positives and negatives.

By design, the obfuscator used in this paper applies obfuscations at compile time to stack and global variables. It does not obfuscate heap variables, even though it would make no difference to Carter. For the selected variables it uses split obfuscations ("split") where it splits to either two, four or eight memory locations. As splits in more than 3 components are rare in practice [11], we limit ourselves to two in the evaluation. To our knowledge, combined split+merge obfuscations that also allow adding spurious memory accesses are not available in any of the obfuscators today. For this reason, we limited the evaluation of the split+merge obfuscation to the simpler cases – without control obfuscation.

Analysis modes

Carter's split variable deobfuscation depends primarily on two things: (a) the value of the link length parameter k, and (b) the number of *additional* memory accesses due to control obfuscation between the accesses to the different components of a split variable. Since parameter k determines how close together the accesses should be in order to classify as candidates, increasing k may lead to more false positives and fewer false

[3] according to D. Wheeler's *sloccount* [35]: `www.dwheeler.com/sloccount`

negatives. Phrased differently, we should use the highest value of k that does not yet incur too many false positives. In the tests, we vary k between two and twelve.

We estimate Carter's sensitivity to spurious memory accesses due to control obfuscations by using the preventive transformation that injects spurious data accesses, as discussed in Section 5. For the split obfuscation, the obfuscator allows us to control exactly the number of additional (data) memory accesses between every two accesses to the components of a split variable. The actual pattern injected by the obfuscator consists of a load, some operations on the data (e.g., an increment), and a store. Carter only cares about the data accesses, so each pattern counts for two accesses. We varied the number of additional memory accesses between two and eight.

The evaluation of Carter's split+merge deobfuscation is limited to the data-flow obfuscation. As we explained above, in this case, we did not have means to insert spurious memory accesses. Similarly to the variable split deobfuscation, we vary k between two and twelve. Both obfuscation modes modified the same variables.

Results of split detection Table 1 shows the result of our deobfuscation of split variables for $k = 6$. It is the simplest possible case, with no further obfuscations.

	Total	TPs	Part.	OA.	FPs	FNs
base64	24	19 (79)	5 (21%)	0 (0%)	0 (0%)	0 (0%)
expr	11	11 (100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
factor	36	22 (61%)	14 (39%)	0 (0%)	4 (1.48%)	0 (0%)
grep	84	74 (88%)	1 (1%)	8 (10%)	6 (0.82%)	1 (1%)
gzip	15	14 (93%)	0 (0%)	0 (0%)	0 (0%)	1 (0%)
lighttpd	175	170 (97%)	4 (2%)	1 (1%)	0 (0%)	0 (0%)
ls	31	29 (94%)	1 (3%)	1 (3%)	0 (0%)	0 (0%)
wget	159	133 (84%)	18 (11%)	1 (1%)	10 (0.63%)	7 (4%)

Table 1. Results for deobfuscation of split variables (k=6).

- **Total in run (`Total`)**: The total number of split variables accessed during the experiment.
- **True Positives (`TP`)**: the variables correctly classified as split.
- **Partial (`Part`)**: Carter correctly identifies the split, but fails to detect all components that make up the variable (e.g., because one part is not really used in calculations) forming a split. While we cannot classify this category as correct, it does provide most of the information required by the cracker.
- **Over-approximated (`OA`)**: Carter correctly identifies the split and all the components, but adds an additional (unrelated) component in the item set. Again, this is not completely correct, but probably quite useful for the attacker.
- **False Positives (`FP`)**: the variables incorrectly classified as split. The percentage represents the rate of erroneously classified variables in the set of all unobfuscated and accessed ones.
- **False Negatives (`FN`)**: Carter did not classify the variables as split, even though it should have.

Figure (3.a) graphically shows the same results for all values of k. We see that in the absence of further obfuscations, Carter is able to detect most of the split variables with low false positives and low false negatives. Moreover, even for high values of k the number of false positives typically remains below 2%.

Next, we evaluate the impact of spurious data memory accesses on our analysis. In principle, we do not know which control obfuscation or preventive transformation is present in the obfuscated binary, and so we do not know the cause of the memory accesses. In our evaluation, we therefore add increasing numbers of spurious memory access between each two accesses to split variables to see what the impact is on our results.

The results are shown in Figures (3.b)-(3.e). They also contain *Expected FNs* (`ExFNs`), i.e., split variables that Carter had no means to identify. If k is smaller than the number of injected accesses, the detection module *cannot* normally detect the split. Observe that we still find variables *occasionally* even for small k and many injected accesses (e.g., for `factor` when $k = 4$ and 6 injected accesses). The reason is that the obfuscator injects instructions between two accesses x_1 and x_2 of a split variable x. It may happen that in the original program two accesses to x occurred close together in logical time. As a result, the accesses to x_2 and x_1 may also still occur close together, in spite of the extra instructions. For instance, assume the program exhibited an access pattern as follows: $x_1 x_2 y z x_1 x_2$. If the obfuscator subsequently injects 6 additional references (A..F), the pattern becomes: $x_1 ABCDEF x_2 ab x_1 ABCDEF x_2$. In this case, the x_2 of the first accesses will still be grouped with the x_1 of the second.

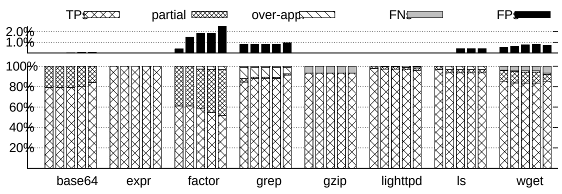
Nevertheless, we conclude that for small values of k, the detection module becomes unreliable as the distance between accesses to the components of a split variable increases. However, we will show in the next section that we cannot keep injecting more memory accesses, unless we are willing to pay a huge penalty in performance.

Finally, many of the false positives in the split variable deobfuscation were cases where the program accessed a two-dimensional array `A` using either one or two subscripts, i.e., `A[x][y]` or `A[i]` where $i = x \times N + y$. Clearly, even these false positives may contain very useful information for a reverse engineer! For instance, in the previous example: if `x` and `y` always access a buffer together, it may suggest a two-dimensional array.

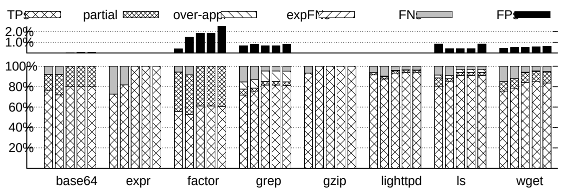
Overhead of preventive transformation Adding spurious memory accesses forces us toward higher values of k. The question is how far we can take this defense. Clearly, adding additional code and memory accesses hurts performance. In this section, we evaluate this cost by running SPECint with and without obfuscations. Specifically, the obfuscator splits the stack variables, after compiler optimizations, of the SPECint applications and we measure the performance relative to non-obfuscated code. Next, it injects increasing numbers of data accesses such that we can we measure their influence. Figure 4 presents the results for the SPECint 2006 benchmark.

We see that the performance really suffers from the additional accesses. The actual slowdown depends on the number of accesses to the obfuscated variables, but may be as high as an order of magnitude. In almost all cases, the slowdown is more than 2x for just 6 injected accesses. We speculate that in many application domains, this would be too high a price to pay.

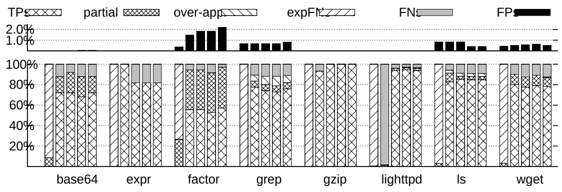
Results of split+merge detection Figure 3f shows the result of our detection of split and merged variables. As we said before, we limit this part of the evaluation to data-



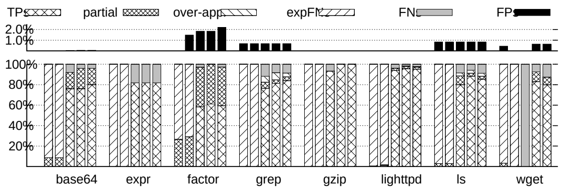
(a) Variable split: no spurious data accesses



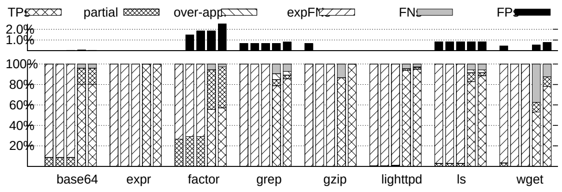
(b) Variable split: 2 spurious data accesses



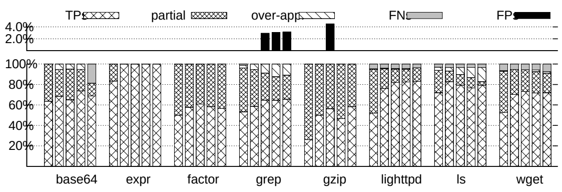
(c) Variable split: 4 spurious data accesses



(d) Variable split: 6 spurious data accesses



(e) Variable split: 8 spurious data accesses



(f) Variable split+merge: no spurious data accesses

Fig. 3. Plots (a)-(e) contain 'variable split' recovery results for $k \in \{4, 6, 8, 10, 12\}$ for N spurious data accesses between the accesses to components of a split variable. Plot (f) contains 'variable split+merge' recovery results for $k \in \{4, 6, 8, 10, 12\}$ with no spurious data accesses. Each value of k is represented by a separate bar. False positives are in a separate plot above the main plot.

flow obfuscation only. In summary, the split+merge detection relaxes the assumptions made by the split detection, to allow the components of merged variables to be accessed separately (refer to Section 4).

The results indicate that the policy for split+merge handles the obfuscation technique successfully, typically detecting more than 50% of the variables perfectly. This percentage is reduced, compared to the split only obfuscation technique, since the relaxed assumptions imply additional uncertainty. This manifests itself as a significant increase in the number of reported partial and over-approximated results. These categories show that Carter successfully identifies the presence of the split components, but does not always precisely infer the boundaries between them.

Finally, the number of false negatives typically remains below 10%, and the number of false positives – below 5%. It means that Carter accurately identifies the obfuscated variables, which is very helpful for the attacker.

Analysis time Running the test suites and analyzing all memory accesses for the appropriate item sets is a fairly labor-intensive operation. Moreover, our current implementation is by no means optimal in terms of performance. Even so, the deobfuscation

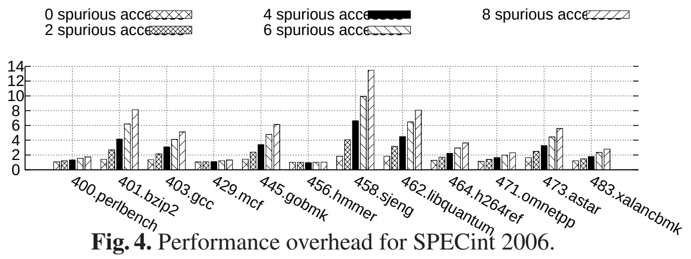


Fig. 4. Performance overhead for SPECint 2006.

procedure is fast enough even for the larger applications. Small applications like *ls* take a few minutes to analyze, larger applications like `wget` take as long as four hours, while SPECint consumes easily twelve hours.

Summary We conclude that for all applications, Carter provides a significant boost when recovering obfuscated variables. Even if the obfuscator spaces accesses to the different split components further apart, Carter still detects the transformation in most cases. If needed, reverse engineers can play with the parameters during the analysis, selecting values that lead to few false positives initially and gradually increasing k. The main message is that for a particular obfuscator, it is relatively straightforward to select good values for these parameters.

7 Application of Carter: binary analysis

To demonstrate the usefulness of Carter, we present the impact of obfuscated variables on the process of reverse engineering. Suppose a reverse engineer is interested in the fd_write function in `wget` and its buffer argument. For illustration purposes, we show the relevant parts of the source code in Fig. (5a). In reality, the reverse engineer has access neither to the source, nor to the debug symbols. In the original code, we see that the buffer argument is sent to $sock\_write$ which in turn calls the underlying $write$ function. Besides being an argument to the $sock\_write$ function, buf is also updated inside the function.

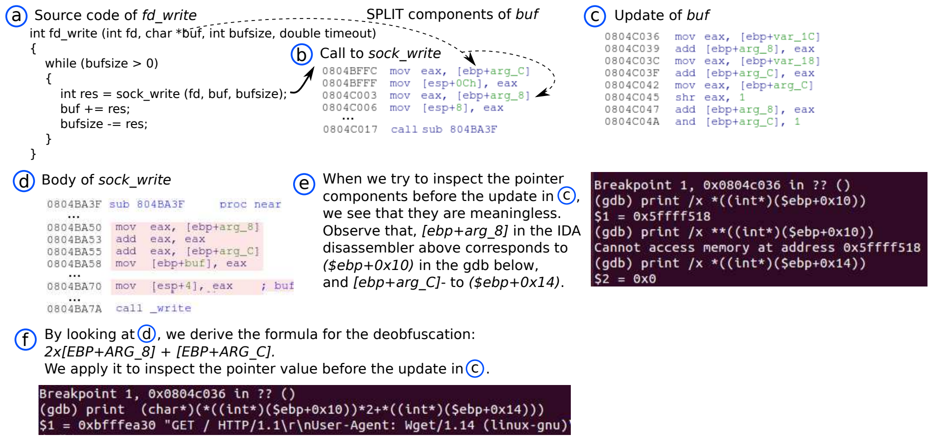
We now strip all debug symbols and apply the obfuscation model of Section 2 to all integer and pointer variables in the binary. Since the obfuscator works interprocedurally, the function arguments will also be split into two components. As a consequence, it will split the buffer argument into the third and fourth argument positions of both fd_write and $sock\_write$—shown as arg_8 and arg_C in the IDA Pro disassembler output in Fig. (5b). Similarly, the update of the variable buf, shown in Fig. (5c), will follow the split rules presented in Section 2.

Now that we have presented the setup of the experiment, let us change our perspective to that of the reverse engineer trying to extract semantics from the stripped binary. Just by looking at the code in Fig. (5a) and (5c), it is impossible to extract semantics for argument positions arg_8 and arg_C, since the buffer pointer is never dereferenced in the code. The reverse engineer is obliged to follow the progress of the argument inside the $sock\_write$ sub-function.

Fig. (5d) shows that the two arguments are combined using arithmetic operations, before being sent to the external $write$ call. Disassemblers can identify that the result of the arithmetic operations has the semantics of the buf argument from the libc prototype.

Intuitively, if a pointer results from the arithmetic combination of two variables, one of them represents a base pointer, and the other the offset. To confirm it, the reverse engineer executes the binary in GDB and checks the value of the two variables just before the update occurs in the original function. The GDB session is presented in Fig. (5e) and shows that one of the variables is a really big integer, but points to invalid memory, while the other has the value of 0. This doesn't correspond to the reverse engineer's intuition about pointer arithmetics at all!

In contrast, by using Carter, the reverse engineer is able to discover the obfuscated memory locations *a priori*. Specifically, Carter presents the reverse engineer with an annotated binary that highlights the possible split locations, making it clear that two memory locations belong together and should be inspected as a group. Moreover, it is now trivial to identify the exact split semantics by checking the (unavoidable) deobfuscation that takes place when the data is used in external library functions–as in Fig. (5d). Using this information, the reverse engineer can now inspect the value of the variable anywhere in the code, by applying the transformation to the given memory locations Fig. (5f).



```
int fd_write (int fd, char *buf, int bufsize, double timeout)
{
   while (bufsize > 0)
   {
      int res = sock_write (fd, buf, bufsize);
      buf += res;
      bufsize -= res;
   }
}
```

```
0804BFFC  mov  eax, [ebp+arg_C]
0804BFFF  mov  [esp+0Ch], eax
0804C003  mov  eax, [ebp+arg_8]
0804C006  mov  [esp+8], eax
   ...
0804C017  call sub 804BA3F
```

```
0804C036  mov eax, [ebp+var_1C]
0804C039  add [ebp+arg_8], eax
0804C03C  mov eax, [ebp+var_18]
0804C03F  add [ebp+arg_C], eax
0804C042  mov eax, [ebp+arg_C]
0804C045  shr eax, 1
0804C047  add [ebp+arg_8], eax
0804C04A  and [ebp+arg_C], 1
```

```
0804BA3F sub 804BA3F     proc near
   ...
0804BA50  mov  eax, [ebp+arg_8]
0804BA53  add  eax, eax
0804BA55  add  eax, [ebp+arg_C]
0804BA58  mov  [ebp+buf], eax
   ...
0804BA70  mov  [esp+4], eax    ; buf
   ...
0804BA7A  call _write
```

When we try to inspect the pointer components before the update in c, we see that they are meaningless. Observe that, *[ebp+arg_8]* in the IDA disassembler above corresponds to *($ebp+0x10)* in the gdb below, and *[ebp+arg_C]*- to *($ebp+0x14)*.

```
Breakpoint 1, 0x0804c036 in ?? ()
(gdb) print /x *((int*)($ebp+0x10))
$1 = 0x5ffff518
(gdb) print /x **((int*)($ebp+0x10))
Cannot access memory at address 0x5ffff518
(gdb) print /x *((int*)($ebp+0x14))
$2 = 0x0
```

By looking at d, we derive the formula for the deobfuscation:
2x[EBP+ARG_8] + [EBP+ARG_C].
We apply it to inspect the pointer value before the update in c.

```
Breakpoint 1, 0x0804c036 in ?? ()
(gdb) print  (char*)(*((int*)($ebp+0x10))*2+*((int*)($ebp+0x14)))
$1 = 0xbfffea30 "GET / HTTP/1.1\r\nUser-Agent: Wget/1.14 (linux-gnu)'
```

Fig. 5. Reverse engineering a binary with split variables.

8 Limitations and recommendations

We have shown that Carter is effective against state-of-the-art data obfuscation techniques, even if they are combined with state-of-the-art control obfuscation. The question we ask in this section is: what can software vendors do to protect their data better? To do so, we suggest measures for obfuscators to increase their potency. Unfortunately, none of them are free and they always increase the cost of a transformation. Worse, few of them appear robust against more advanced deobfuscators.

Carter detects split variables by selecting memory locations that (1) are accessed "together", and (2) exchange data. It will be difficult for an obfuscator to avoid the data exchange altogether unless the whole environment is aware of the transformation. The only (intriguing) solution we can think of are covert channels that hide the information exchange from the DIFT module (Section 3.4). Covert channels would significantly increase the complexity of the obfuscation.

A less radical direction is to increase the distance between the accesses to the components of a split variable—in an attempt to exceed the link length parameter k—just like we did when we injected spurious memory accesses. We have already seen that doing so is expensive due to the extra memory accesses (and the corresponding reduction of locality of reference). We also saw that the results are limited as the adversary can increase k, at the cost of some more false positives and negatives.

The best way to increase the distance while reducing the overhead is to make only certain accesses distant by means of instructions. However, even in this case adversaries may benefit from Carter's analysis by relaxing the requirement that the variables need to be *always* accessed together—again at the cost of additional false positives.

Finally, as Carter looks for variables with the same allocation time, it would be advisable to give components of a split variable different allocation times. For instance, by allocating one part as a static variable in the data segment, and another in the function frame. Doing so requires Carter to relax another one of its constraints. Again, the reduction of locality would probably lead to additional overhead (due to cache and TLB pollution). Also, it does not invalidate the method, but makes it less precise.

9 Related Work

Program obfuscation is a mature field. Many commercial obfuscators work by transforming source code. Examples include Stunnix [32] and Semantic Designs' framework [28]. However, software developers may also opt for compiler-driven obfuscation like Morpher [26] and CodeMorph [31], or even the multi-layer defense offered by Irdeto's Cloakware [19].

Perfect obfuscation is impossible in general [4], but practical reverse engineering of obfuscated code is still difficult. To the best of our knowledge, all existing work on deobfuscation targets code, rather than data obfuscation. To illustrate this, we briefly review existing work on deobfuscation of compiled code.

Most of the work on obfuscation, like [23, 36], strives for resistance against static analysis. The authors do not try to defend against the use of non-conservative, (partially) automated, dynamic analyses. For a long time, the same was true for attackers, but Madou et al. [24] illustrate the potential of hybrid static-dynamic attacks through a case study of an algorithm for software watermarking [8].

A popular branch of code deobfuscation is concerned with recovering the sequences of instructions intended by a programmer. Kruegel et al. [21] present an analysis to disassemble an obfuscated binary. Lakhotia et al. [22] apply stack shape analysis to spot when an obfuscated binary makes library calls even if it does not use the `call` and `ret` instructions. Finally, Udupa et al. [33] examine the resilience of the control flow flattening obfuscation technique [34, 7] against attacks based on combinations of static and dynamic analyses.

Opaque predicates also attracted much research. The simplest method to break them is dynamic analysis. However, due to the code coverage problem, it does not always provide complete or reliable solutions. Madou et al. [24] propose a hybrid static-dynamic mechanism. They statically identify basic blocks that contain opaque predicates, and dynamically execute them on all possible inputs. Some obfuscators [11, 12, 25] hinder this approach by tricking the program into returning an artificially large slice to be analyzed. Dalla Preda et al. [14] present an abstract interpretation-based methodology for removing certain types of opaque predicates from programs. None of these solutions solve the problem in general.

Metasm [17] is a framework to assist a reverse engineer by disassembling a binary, and building its control flow graph, even in the presence of control obfuscation. Saidi et al. [27] developed an IDA Pro plugin to help deobfuscate malware instances. The tool tackles a few categories of obfuscations, e.g., malware packing, anti analysis techniques, and Windows API obfuscation.

To deal with advanced control obfuscations like virtualization, Coogan et al. [13] identify instructions that interact with the system by system calls. Next, they determine which instructions affect this interaction. The resulting set of instructions is an approximation of the original code. Sharif et al. [29] also target virtualized malware and record a full execution trace and dynamic taint and data flow analysis to identify data buffers containing the bytecode program, so they can reconstruct the control flow graph.

Anckaert and Ceccato worked on the evaluation of obfuscating transformations [2, 5]. They assess both code metrics, such as the computational complexity of static analysis, and the difficulty of understanding the obfuscated code by human analysts.

The most important outcome of our literature study, is that there is, to our knowledge, no work on the recovery of obfuscated data.

10 Conclusion

In this paper, we evaluated the strength of data obfuscation techniques. In our evaluation, we included common and powerful techniques: splitting, and splitting and merging variables over multiple memory locations. We showed that dynamic analysis of memory access patterns is a useful way for semi-automated deobfuscation of the data. With false positive rates below 5%, and false negative rates typically below 10%, a determined cracker can successfully use them to recover the original data. We conclude that the obfuscations are at least vulnerable. So much so, that we believe that the data obfuscations examined in this paper should no longer be considered safe. Finally, we have shown that we can raise the bar for crackers by taking additional measures, but we doubt that these measures will be safe in the long run.

Acknowledgment

This work was partially funded by the European Research Council through project ERC-2010-StG 259108-ROSETTA, the EU FP7 SysSec Network of Excellence and the NWO CyberSecurity project ”Re-Cover” (628.001.005).

References

1. Codevirtualizer: Total obfuscations against reverse engineering. `http://oreans.com/codevirtualizer.php`, 2008.
2. B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In Proc. of the 2007 ACM workshop on Quality of protection, QoP ’07, 2007.
3. S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In Proc. of the 6th ACM Conference on Computer and Communications Security, 1999.
4. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In Proc. of the 21st Annual International Cryptology Conference on Advances in Cryptology, CRYPTO’01, 2001.
5. M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. Towards experimental evaluation of code obfuscation techniques. In Proc. of the 4th ACM workshop on Quality of protection, 2008.
6. V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in vivo multi-path analysis of software systems. In 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011.
7. S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In Proc. of the 4th International Conference on Information Security, ISC’01, 2001.
8. C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In Proc. of the ACM SIGPLAN 2004 conference on Programming language design and implementation, PLDI’04, 2004.
9. C. Collberg and J. Nagra. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. 2009.
10. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Sciences, The University of Auckland, Auckland, New Zealand, 1997.
11. C. Collberg, C. Thomborson, and D. Low. Breaking Abstractions and Unstructuring Data Structures. In Proc. of IEEE International Conference on Computer Languages, ICCL’98, 1998.
12. C. Collberg, C. Thomborson, and D. Low. Obfuscation techniques for enhancing software security, 2003.
13. K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In Proc. of the 18th ACM conference on Computer and communications security, CCS ’11, 2011.
14. M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque predicates detection by abstract interpretation. Algebraic Methodology and Software Technology, LNCS 4019, 2006.
15. C. Ding and K. Kennedy. Inter-array Data Regrouping. In Proc. of the 12th International Workshop on Languages and Compilers for Parallel Computing, LCPC’99, 1999.

16. C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In Proc. of the ACM SIGPLAN 2003 conference on Programming language design and implementation, PLDI'03, 2003.
17. Y. Guillot and A. Gazet. Automatic binary deobfuscation. Journal in Computer Virology, 2010.
18. Intel. Pin - A Dynamic Binary Instrumentation Tool. `http://www.pintool.org/`, 2011.
19. Irdeto. Application security. `http://irdeto.com/en/application-security.html`.
20. V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In Proc. of the 8th Annual International Conference on Virtual Execution Environments, VEE'12, 2012.
21. C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In Proc. of the 13th conference on USENIX Security Symposium, SSYM'04, 2004.
22. A. Lakhotia and E. Uday. Stack shape analysis to detect obfuscated calls in binaries. In In Proc. of 4th IEEE International Workshop on Source Code Analysis and Manipulation, 2004.
23. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In Proc. of the 10th ACM conference on Computer and communications security, CCS'03, 2003.
24. M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In Proc. of the 5th ACM workshop on Digital rights management, DRM'05, 2005.
25. A. Majumdar, S. J. Drape, and C. D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In Proc. of the 2007 ACM workshop on Digital Rights Management, DRM'07, 2007.
26. Morpher. Software protection service. `http://www.morpher.com/`.
27. H. Saidi, P. Porrass, and V. Yegneswaran. Experiences in malware binary deobfuscation. In the 20th Virus Bulletin International Conference, 2010.
28. Semantic Designs. C source code obfuscator. `http://www.semdesigns.com/products/obfuscators/CObfuscator.html`.
29. M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In Proc. of the 2009 30th IEEE Symposium on Security and Privacy, 2009.
30. A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In Proc. of the 18th Annual Network & Distributed System Security Symposium, NDSS'11, 2011.
31. SourceFormatX. Codemorph source code obfuscator. `http://www.sourceformat.com/code-obfuscator.htm`.
32. Stunnix. `http://stunnix.com/`.
33. S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In Proc. of the 12th Working Conference on Reverse Engineering, WCRE'05, 2005.
34. C. Wang, J. Hill, J. C. Knight, and J. W. Davidson. Protection of software-based survivability mechanisms. In Proc. of the 2001 International Conference on Dependable Systems and Networks, DSN'01, 2001.
35. D. A. Wheeler. Sloccount. `http://www.dwheeler.com/sloccount/`.
36. Z. Wu, S. Gianvecchio, M. Xie, and H. Wang. Mimimorphism: a new approach to binary code obfuscation. In Proc. of the 17th ACM conference on Computer and communications security, CCS '10, 2010.
37. Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In Proc. of the ACM SIGPLAN 2004 conference on Programming language design and implementation, PLDI'04, 2004.