Bit-Level Taint Analysis

Babak Yadegari
University of Arizona
Computer Science Department
babaky@cs.arizona.edu

Saumya Debray
University of Arizona
Computer Science Department
debray@cs.arizona.edu

***Abstract*— Taint analysis has a wide variety of applications in software analysis, making the precision of taint analysis an important consideration. Current taint analysis algorithms, including previous work on bit-precise taint analyses, suffer from shortcomings that can lead to significant loss of precision (under/over tainting) in some situations. This paper discusses these limitations of existing taint analysis algorithms, shows how they can lead to imprecise taint propagation, and proposes a generalization of current bit-level taint analysis techniques to address these problems and improve their precision. Experiments using a deobfuscation tool indicate that our enhanced taint analysis algorithm leads to significant improvements in the quality of deobfuscation.**

I. Introduction

Dynamic taint analysis tracks the flow of data through a program's execution and marks all data derived from certain sources of interest (e.g., user input). There is a wide body of literature on taint analysis (e.g., see Schwartz *et al.* [21]). The key idea is to maintain and propagate meta-data (the "taint") alongside the program's computation such that values derived from a source of interest are flagged as tainted. Most taint analyses maintain one taint bit per byte or word of program data, though some researchers have discussed finer-grained analyses [7], [28]. Taint analysis is used in a variety of applications, including application security [10], [15], [16], [19], software testing and debugging [12], malware analysis [13], [31], and deobfuscation of obfuscated code [23], [29].

The extensive use of taint-based security analyses has led to attacks aimed at defeating such analyses [2]. Broadly speaking, there are two such kinds of attacks. In the first kind of attack, the attacker wants to control the contents of some locations (e.g., a return address or function pointer) and the defender uses taint analysis to detect whether this happens. Such attacks typically rely on implicit information flows to induce *under-tainting* or false negatives, i.e., cause taint analyses to infer that certain values are not tainted even though they are influenced by tainted data [2]. In the second kind of attack, the attacker wants to conceal the functionality of some code (e.g., a malware attack payload) by obfuscating the code, and the defender uses taint analysis to track the flow of values through the code and infer its functionality. Such attacks typically rely on *over-tainting* or false positives, i.e., cause taint to flow to so much of of the program that various unrelated computations cannot be teased apart. This paper is concerned with the second kind of attack/imprecision, namely, over-tainting.

This paper makes two main contributions. The first is to discuss some sources of over-tainting in traditional taint analyses. The second is to describe an enhanced bit-level taint analysis that addresses these problems and thereby significantly improve the precision of taint analysis. We discuss the application of our ideas in the context of a tool for automatic deobfuscation of obfuscated binaries, focusing in particular on code obfuscations that target the flow of values through a computation [4] [6]. These obfuscations can intermix bits from different program values—in essence, shuffling the bits from the values of a number of unrelated program variables such that, after the shuffle, each byte contains bits from multiple variables and each variable's bits are distributed across multiple different bytes. In such cases, standard taint analyses (even fine grained bit-precise ones) can suffer a considerable loss of precision due to over-tainting; this, in turn, can significantly degrade the quality of the deobfuscated code.

We address these problems via a precise *dynamic taint analysis* algorithm that extends the basic taint analysis algorithm in two ways. First, we use a fine-grained bit-level analysis, with precise mapping functions to model the taint effects of different operations. For example, the effects of an *add* operation are modeled differently than those of an *xor* operation. Second, we distinguish between, and keep track of, different taint sources. This makes possible a more refined treatment of taint propagation and allows us to avoid certain kinds of *taint explosion*. The precision of the taint analysis depends on these two parameters; we note that different applications of taint analysis will, in general, require different levels of precision, and in some cases it may suffice to consider only a subset of taint sources rather than all of them.

The remainder of this paper is organized as follows. Section II gives some background on taint analysis and briefly discusses some evasion techniques used to make the analysis imprecise. Section III gives the details of our method for dynamic taint analysis. Section IV evaluates different taint analysis algorithms including our method with a few emulated test programs. Section V discusses related works in the area followed by conclusions on Section VI.

II. Background and Motivation

Taint analysis was originally designed as a way to track the flow of data through a computer system, potentially including both application program state (e.g., program variables and memory) and system state (e.g. operating system state, files).

```
1   int a, b, c
2   a = read()
3   b = ~a
4   c = (a & b)
5   if (c == 0){
6       // True
7   } else {
8       // False
9   }
```

Fig. 1: An example of taint propagation

This makes it possible to detect runtime violations of security policies, e.g., where an externally-supplied or user-controlled value is written to a security-sensitive memory location [10], [13], [15], [16], [19], [31]. More recently, taint analysis has been used to reason about and simplify obfuscated code [23], [29]. The important role played by taint analysis in these applications means that it is important that the implementation is precise and resilient against attacks and evasion techniques.

There are two kinds of imprecision in taint analysis: *over-tainting* and *under-tainting*. Over-tainting occurs when code or data identified by the analysis as tainted is not in fact influenced by any taint source. Under-tainting occurs when code or data that is influenced by a taint source is not identified by the analysis as tainted. Such imprecision can be problematic, especially in systems where the result of the taint analysis is critically important. For example, if taint analysis is used to detect leakage of sensitive information, under-tainting means that the system has failed to ensure the privacy of the sensitive data; in a policy enforcement system using taint analysis, over-tainting can lead to disruptive false alarms. There have not been enough studies on limitations of the taint analysis and in particular, how to improve the precision of the current existing taint analysis techniques. Researchers (e.g. [2], [20], [22], [30]) have raised concerns about some specific shortcomings of taint analysis and how these limitations could be used to attack the analysis. Neither the mentioned studies nor any previous work, as far as we are aware of, have studied the effects of code obfuscation on the precision of the taint and in general data-flow analysis or how to address them.

Standard taint analyses, performed on ordinary compiler-generated code, usually yield an acceptable level of precision since the code has normal data and control flow. However, this may not hold true for all binaries. For example, code obfuscation can transform normal code/data flows of a program to make analysis difficult. The resulting obfuscated control/data flow can lead to a loss of precision in taint analysis. Figure 1 shows a simple example where the standard taint analysis over-taints. First, three variables are defined at line 1 and variable `a` is assigned a value read from input at line 2. We start tainting by *introducing* taint to variable `a` and propagate it through the code snippet. Since `a` is tainted, variables `b` and `c` become tainted at line 3 and 4 respectively, leaving both `a` and `b` tainted. Variable `c` being tainted, makes the `if` statement tainted at line 5 and to any dynamic or static analysis, the behavior of the program at line 5 is not predictable. Looking at the example more carefully, variable `b` contains the negation of `a` so the result of `AND`-ing these two variables will always be zero regardless of the value of `a`. Since the value of `c` does not depend on data from any taint source, it should not considered to be tainted. However, standard taint analysis will identify `c` as tainted because the operands of the operation defining `c` (line 4) are tainted. The problem here is that the standard algorithm does not propagate enough information to determine the taint status of `c` precisely.

Figure 2 shows other examples where conventional assumptions about flow of data may not hold. Figure 2(a) shows an example of *split variables* obfuscation [4] and illustrates the over-tainting effects that arise when the taint analysis is not sufficiently fine-grained. If we start tainting by marking variable `a` at line 4 and propagate taint at byte or at higher levels of granularity, variables w_1 and w_2 get tainted at lines 6 and 7 respectively, since `a` is tainted. Variable `b` also gets tainted at line `i+1` because w_1 and w_2 were previously tainted. However, notice that only the even bits of w_1 are tainted, since its odd bits come from `b`; similarly, only the odd bits of w_2 are tainted. Thus, when `b` is being retrieved at line `i+1`, only even bits of w_1 and odd bits of w_2 are used, so `b` should not be not tainted. Figure 2(b) shows a snippet of obfuscated code generated by a commercial obfuscation tool called *EXECryptor* [24]. This code snippet, rephrased from assembly language for ease of understanding, shows an initial segment of a long sequence that involves thousands of instructions; the full sequence is omitted due to space constraints. This piece of code actually shows the conditional control transfer mechanism that can be implemented in a virtual machine by examining the appropriate bit of the actual `psw flags` of the cpu. Line 1 of the code retrieves the `EFLAGS` register of the cpu.[1] The rest of the code involves a long sequence of arithmetic operations on the flags and finally at line `i` the flag is being tested to whether carry out the control transfer or not. This kind of obfuscation causes the taint analysis to over-taint and leads to *taint explosion*.

III. Our Approach

A. Overview

In general, taint analyses consist of three main components: *taint markings*, *mapping functions* and, *granularity*. This discussion focuses on how different choices for these components affects the precision of taint analysis. We discuss different factors that impact the precision of analysis as well as some of the techniques used to defeat taint analysis based on the characteristics of these components.

a) Taint Markings: Taint markings refer to the information needed to be kept for annotating and marking the relevant code that either affects the data or involves propagating it. Typically ***T*** is used to denote if the data/code is tainted and ***F*** for the data/code that is not tainted. Clearly this binary representation is only able to determine if something is tainted

[1]The corresponding instruction sequence in `x86` assembly is `pushfd` followed by a `pop` *reg*; in this case *reg* is `edx`.

```
1    char a, b, w1, w2
2    char odd_bits = 01010101b
3    char even_bits = 10101010b
4    a = read()
5    b = 10
6    w1 = (a ∧ even_bits)
          ∨ (b ∧ odd_bits)
7    w2 = (a ∧ odd_bits)
          ∨ (b ∧ even_bits)
          ...
i    a = (w1 ∧ even_bits)
          ∨ (w2 ∧ odd_bits)
i+1  b = (w1 ∧ odd_bits)
          ∨ (w2 ∧ even_bits)
i+2  print(b)
```

(a) Variable splitting

```
1    edx = flags
2    eax = 0x6b30f626
3    edx = edx ∨ 0xfffff73e
4    eax = edx ∧ eax
5    edx = eax
6    eax = rotate_eight(eax, 0x10)
7    dx = rotate_left(dx, 0x3)
8    edx = edx + 0x6c7caf05
9    edx = edx ⊕ 0xe0395c49
10   ax = ax ⊕ dx
11   edx = eax
          ...
i    if((eax ∧ 0xe0000000) == 1){
i+1       True branch
i+2  } else {
i+3       False branch
i+4  }
```

(b) An example code of EXECryptor

Fig. 2: Code examples where standard taint analysis over-taints

or not and so only one bit is sufficient to keep track of a tainted entity where the size of this entity can be a bit, byte, word or a double-word. While simple and efficient, this puts a big limitation on taint analysis. By the amount of information that can be inferred from one bit, it is not possible to determine from which taint source a tainted unit is derived. This limits the ability of the analysis to reason about the effects of various kinds of arithmetic and logic operations in the language on tainted data. For example the result of xor-ing two bits will be marked as tainted if either input is tainted, however if both bits are from the same taint source the result will always be zero regardless of the actual values of the tainted bits, i.e., unaffected by the tainted input.

Moreover, we observe that some obfuscations can intermix bits from different program values—in essence, shuffling the bits from the values of a number of unrelated program variables such that, after the shuffle, each byte contains bits from multiple variables and each variable's bits are distributed across multiple different bytes; Collberg and Thomborson refer to this obfuscating transformation as *split variables* [4]. In such cases it is not enough to only mark a bit as tainted or untainted: we have to also keep track of each source of the taint individually at the bit-level. In order to maintain a balance between precision and performance, our current implementation maintains bit-level taint-source tracking only on condition code flags as we have found it gives us enough amount of accuracy to deal with obfuscations but it is relatively simple to extend the idea to any kind of taint source.

An even more challenging code obfuscation technique for taint analysis to deal with is opaque predicates or opaque variables [1], [14]. An opaque variable is a variable which at some point in the program has some property that is known to the obfuscator but it is difficult by the analysis to infer this property. Similarly, an opaque predicate is a predicate in which the outcome of its evaluation is known to the obfuscator. These kinds of obfuscations can cause the analysis to lose precision by faking the data-flow. One can imagine scenarios where an opaque variable seems to be participating in carrying taint but in fact they carry a constant value.

As an example, let us go back again to our code snippet in Figure 1. It can be shown that by using different taint markings we can prevent the analysis from over-tainting at line 4. We start tainting by introducing taint to the `input` at line 2, we mark the `input` with taint mark T_1. `input` then is copied to `a`, so `a` gets the taint mark T_1 as well. At line 3, `b` gets the negation of `a` and since it is not equivalent to `a` anymore, we need to use a different mark to annotate `b`. To mark `b` as tainted, we use the marking T_2 which in fact has the value of $\overline{T_1}$, resulting from applying the negation operator to the markings of variable `a`. The relation between various taint markings are defined in the *Mapping functions* of the next subsection so the details are skipped here. At line 4, `c` is the result of logical `and` of variables `a` and `b`, but the markings tell us that `b` is the negation of `a` and hence the result of the `and` is always zero, leading us to conclusion that `c` is not tainted. Having only a binary representation to track the taint does not give us enough clues about the outcome of the operations on tainted sources, so for a precise result we need to keep distinct markings for different sources.

b) Mapping Functions: Mapping functions or *propagation policies* define how the taint markings of the source operands of a statement `s` are propagated to its destination. In standard taint analysis, the destination operand is typically marked as tainted if any of the source operands is tainted regardless of how the specific semantics of `s` affects its destination operands. This conservative approach can be imprecise and lead to over-tainting. Note that changing the granularity of the analysis does not help: even at bit-level granularity, a conservative mapping mechanism causes the destination to become tainted if any source operand is tainted, thereby giving the same result as performing the analysis at coarser levels of granularity. Moreover, sometimes only a part of the destination gets tainted. For example, performing logical shift on a tainted operand will leave a portion of the result untainted while the

simple union based mapping function will mark the whole destination tainted. It is important to have mapping functions that are somehow equivalent to the functional semantics of the corresponding statement in the language, because the way taint propagates in a statement from its sources to its destinations correlates with the functional semantics of the statement.

Returning to the example in Figure 1, the mapping functions for assignments at line 1 simply copies the markings of the source, which is `input`, to destination variable `a`, so if `input` is marked with T_1, T_1 is copied to `a`'s markings. This is actually consistent with the functional semantics of the assignment, since it copies the source to destination without any modifications. For the negation operation at line 3, we can use any different marking from T_1, but it makes sense to use $\overline{T_1}$ which is the result of applying the negation operation on T_1. At line 4 we need to compute the taint for `c` which is the result of a logical and on `a` and `b` variables. It again makes sense to use logical `and` as the mapping function for this operation. Taking logical and of T_1 and T_2, since T_2=$\overline{T_1}$, leaves variable `c` untainted, which is what we expect in this case. This is actually important to note that to fully benefit from using different mapping functions, we need to have distinct markings for different taint sources because using the functional semantics of statements to map the taints otherwise will cause other types of imprecision.

c) Granularity: Another important factor that affects the accuracy of the taint analysis is the granularity that the analysis is performed on. This could vary anywhere from word-level to bit-level depending on the application and the domain where the analysis is used. This simply specifies how much of the data can be represented by a bit of taint. For example with word-level granularity, one bit of data is enough to mark any word in the program to determine whether it is tainted or not. Clearly the more finer-granular the analysis is, the more accurate the result will be, i.e., doing the analysis at bit-level will give the best result. Sometimes it is necessary to perform the analysis at bit-level where the sources of taint are single bits, e.g., the `eflags` register where every single bit carries important information individually when the flags are affected by an operation with tainted sources or some of the bits may not even be tainted depending on the operation causing flags to be tinted, so marking the whole `eflags` register with one marking will introduce unnecessary imprecision to the analysis.

It turns out that traditional byte- or word-level taint analysis is too imprecise for our needs and can result in significant over-tainting. To address this problem, we turn to a finer-grained bit-level taint analysis: instead of having a taint mark for each byte *b*, we associate each bit with a taint mark. The final algorithm actually might use a mixture of different sized units for different variables. For example, if a variable is only accessed via byte-sized reads or writes, it suffices to associate each byte of the variable with a (possibly distinct) taint mark. However for single-bit variables like control flags of the CPU, it is better to work at bit-level granularity and use distinct taint marks for each control bit.

All the above factors have significant impact on the overall precision of the analysis and changing one of the factors has significant effects on the other factors. For example changing the granularity level from byte to bit will need new mapping functions that reflect the semantics of the operations at bit-level instead of byte-level but this does not change the functional semantics of the mappings. Moreover, taint markings should change so that one taint mark instead of being able to address a byte of data, it will be able to cover every single bit of data.

B. Algorithm

Our algorithm is conceptually analogous to the standard taint analysis algorithm but it tries to address the shortcomings of the taint analysis discussed in this paper. Algorithm 1 gives a very high level overview of the proposed taint analysis algorithm which actually implements the mentioned characteristics of the analysis and the steps to perform these tasks will be discussed in detail.

Algorithm 1: Taint Analysis Algorithm

Input: an execution trace T
Result: annotated trace T'

```
1 T' ← IdentifyTaintSources(T)
2 TaintedVars ← InitializeTaintedVars(T')
3 Markings ← CreateMarkings(T')
4 s = T(0)
5 while s ← T'.NextInstruction()! = ∅ do
6 |   T' ← Annotate(Markings, s)
7 |   TaintedVars, Markings ← Map(Markings, s)
8 end
```

1) Identifying Taint Sources: As shown in Algorithm 1, identifying sources of taints or *introducing taint* is the first step of the analysis. Our implementation currently considers the output of all system calls that a program makes as a source of taint, but taint can be introduced by any variable or value. Defining the taint sources to be the output of system calls gives us the flexibility to filter the inputs to a program that are interesting to the analyst. For example if the analyst is only interested in the flow of data that a program sends or receives over the network, he/she can restrict the input functions to be of those communicating with the network and skip the others. Once the sources of taints were identified, we need to propagate this taint through the program.

2) Taint Markings: The algorithm continues by defining the markings at the beginning. By our definition, every bit of a tainted source can be a distinct taint marking. There might be a situation where it is not feasible to have a distinct marking for every single bit of tainted sources, for instance when program processes an infinite incoming stream of data, so there is a trade off between precision and performance of the analysis. As mentioned before, to achieve the best result with acceptable performance, one might choose a mixed model of taint markings. For those variables of bigger size–like byte- or word-sized variables–a taint marking is chosen as large as

the size of the variable and for those of finer-grained sizes, taint markings could be assigned to each bit. For example for a byte in memory, one taint marking assigned to the whole byte would be enough but for control flag bits, one taint mark assigned to each flag bit is needed. The initial set of taint markings are defined after identifying the sources and as the algorithm proceeds, it keeps updating this set of markings by either adding new markings or removing ones that no longer exist. Moreover, some of the sources can have the same markings. For instance if a program reads a file and writes it to another file without doing any interesting computations on the data, there is no significance in distinguishing between the markings of the data.

3) Mapping Functions: In algorithm 1, mapping functions are closely related to functional semantics of the underlying statements. The `Map` function in Algorithm 1 takes the current taint markings and based on the statement `s` and its operands, whether they are constants or not, applies the appropriate mappings from the marked sources (if any) to the affected data or destination operands. The `Map` function can be thought of as a function that *emulates* the effects of statement `s` on the taint marks of the operands. By emulate we mean that this is not always as simple as executing the statement `s` on taint markings of the operands and some operations need modifications to be precise. To make the behavior of the mapping functions as general as possible (in which they cover all the operations), we try to define them in terms of `x86` machine language.

Generally `x86` instructions can be divided into three major categories: *Data handling and memory operations*, *Arithmetic and logic operations* and *Control flow operations*. We define three major categories of mapping functions, corresponding to these three categories, as follows:

- *Data handling and memory operations*: Most of instructions in this category are involved in copying data between sources like registers and/or memory. The behavior of the operations on this category is not complicated semantically compared to arithmetic and operations provided by the machine. Similarly it suffices for a mapping function of a data movement operation to map the taint marks of the source operand(s) to the destination operand(s).
- *Arithmetic and logic operations*: The operations on this category are the main source of precision loss in standard taint analysis. Failing to propagate the taint precisely for an arithmetic operation will soon cause over-taint if it is followed by similar operations. A mapping function for an arithmetic operation can be thought of emulating the arithmetic operation on the taint marks of the statement based on the semantic evaluation of the statement and its operands.
- *Control flow operations*: These operations are not explicitly involved in data-flow. Since this study focuses on explicit data flow, we do not discuss the effects of control transfers on the taint analysis. The analysis of implicit information flow through control transfers has been studied in detail elsewhere (e.g. [3], [9]) and the solutions proposed there can be adopted in our work.

Algorithm 2: Map function

```
1  Procedure Map(Markings, s)
3      src ← IdentifySources(s)
5      dst ← IdentifyDestinations(s)
7      switch s do
8          case s ∈ Data handling and memory operations
9              dst.markings ← src.markings
10         end
11         case s ∈ Arithmetic and logic operations
12             dst.markings =
               ArithmeticMap(s, src, src.markings)
13         end
14         otherwise
15             Continue
16         end
17     endsw
19     return
```

Algorithm 2 describes the mapping functions in pseudo code. Mapping functions are the main component in propagating the taint through the program. As mentioned earlier, how the taint markings are defined in the algorithm determines the behavior of the mapping function. Let's make it clear that the significance of defining mapping functions bases on the functional semantics of the statements is when we keep taint markings separate for different taint sources, otherwise applying functional semantics on the same taint markings introduces even more imprecision to the analysis. So only for the sake of discussion we assume that the analysis keeps taint sources distinct. The algorithm starts by identifying sources and destinations of the statement at lines 2 and 3. Every instruction or statement is a transformation of the source operands to its destination operands, hence by identifying source and destination operands, we want to apply the appropriate transformation from the source to the destination. The transformation for data handling statements or memory operations is simply making a copy of the source to the destination operand, which is what happens at line 9 of Algorithm 2 for data movement and handling operations. An example of a data handling operation is the `pop eax` instruction in `x86` assembly language. This instruction copies the memory location on top of the stack, pointed to by `esp` register, to the `eax` register. For this instruction, the source operand is the memory location on top of stack and the destination is `eax` register so to propagate the taint for this instruction, taint markings of the source memory location should be copied to the destination register.

The transformation however is different and more complicated for arithmetic and logic operations and depends specifically on the functional semantics of the underlying operation. The functional semantics of each operation is a function of the operation and the operands. For many of the

arithmetic operations, applying the corresponding arithmetic or logic operation on the source operand marks and them as taint marks for destination operands suffices, but there are exceptions where the transformation should be done carefully. Figure 3 lists pseudo code of the mapping procedure for two arithmetic operations: `add` and `xor`. Each operation in these examples takes two source operands, $\texttt{src}_1$ and $\texttt{src}_2$, and stores the result in `dst` operand.

```
foreach (bit i of srcs):
  if (src1[i] is constant):
    if (src1[i] is 1):
      dst[i].t = ¬src2[i].t
    else:
      dst[i].t = src2[i].t
  elif (src2[i] is constant):
    if (src2[i] is 1):
      dst[i].t = ¬src1[i].t
    else:
      dst[i].t = src1[i].t
  else:
    dst[i].t =
      scr1[i].t ⊕ src2[i].t
```

(b) `xor`

```
foreach (bit i of srcs):
  if (src1[i] is constant):
    if (src1[i] is 1):
      dst[i].t = src2[i].t
      dst[i+1].t = src2[i].t
    else:
      dst[i].t = src2[i].t
  elif (src2[i] is constant):
    if (src2[i] is 1):
      dst[i].t = src1[i].t
      dst[i+1].t = src1[i].t
    else:
      dst[i].t = src1[i].t
  else:
    dst[i].t =
      scr1[i].t + src2[i].t
```

(a) `add`

Fig. 3: Mapping functions for `add` and `xor` operations, each storing the result of the operation in `dst` with sources `src1` and `src2`

As mentioned before, the behavior of the mapping depends on the operands and based on the inputs to the statement `s` there are different cases which should be dealt with differently. Here we are assuming that at least one of the operands is tainted, otherwise the instruction is not involved in taint propagation so we do not discuss them.[2] These cases are:

1) If one operand is tainted and the other is constant, the markings on the result of the operation are obtained by applying the semantics of the operation to the constant operand and the markings bits. For example, as shown in Figure 3(a), for `xor` binary operation when one of the inputs is tainted with taint mark $\texttt{T}_1$ and the corresponding bit of the second operand is constant, the resulting bit will be marked with $\texttt{T}_1$ if the constant bit is `0` because xor-ing a bit `b` with `0` produces the same bit `b`. Similarly, if the constant bit is `1`, then the resulting bit gets $\overline{\texttt{T}_1}$ (the complement of $\texttt{T}_1$) since xor-ing a bit `b` with `1` flips `b`.
2) The other case is when both bits are tainted, then depending on the operation the result is either a new tainting or not tainted. Again as shown in Figure 3 if both operands have taint markings $\texttt{T}_1$ this means that they are from the same taint source and so the result of xor is `0`, or one bit is $\texttt{T}_1$ and the other is $\overline{\texttt{T}_1}$ where the xor result is 1 and hence the result is not tainted, or the markings are $\texttt{T}_1$ and $\texttt{T}_2$ where the result of xor is unknown so the corresponding bit can be marked with a new taint marking.

Another interesting example of mapping functions is the add operation which stores the sum of its two operands in the destination operand. The pseudo-code for this operation is also given in Figure 3(b). Similar to the `xor` operation, there are two cases to consider:

1) First where one of the source operands is tainted and the other one contains a constant. Intuitively, adding `0` to a bit does not have any effect on the result. Similarly adding `1`, depending on the summand bit, produces a carry if the other bit is one and not otherwise. In case of adding a tainted bit with taint mark $\texttt{T}_1$ and a constant bit, since we do not know what the tainted bit is, there are different cases. If the constant bit is `0` then the result is the same as the tainted bit so the resulting bit gets the taint mark $\texttt{T}_1$. Otherwise if the constant bit is `1`, the addition may produce a carry so the next bit that receives the carry should also be tainted hence both bits get markings $\texttt{T}_1$.
2) Second where both operands are tainted, then we can sum the markings of operands. For example the taint mark of adding two tainted bits with marks $\texttt{T}_1$ and $\texttt{T}_1$ would be $\texttt{T}_1$ + $\texttt{T}_2$. The advantage here again is if $\texttt{T}_2$=$\overline{\texttt{T}_1}$, then we are adding a bit with its complement so the result should be `0` and not tainted as is the case here.

We cannot always infer the taint markings of the result by simply applying the operation on the sources. For instance if the arithmetic operation is shift right/left, then the functional semantics of this operation would be to shift right or left the operand while the amount of shift is determined by another operand. If the second operand is constant, we can shift the taint marks by the amount determined by the constant value, but if the second operand is also tainted, then the functional semantics of the operation depends on some tainted value and can not be determined. In this situation the mapping function will simply apply the union function on the markings of the source operands and the destination gets the new taint marks. There are also single source arithmetic operations like `not` operation. For these operations we can simply apply the semantics of the operation on the taint markings of the source and store the result in the destination operand.

C. Implementation

The focus of this research study is not to produce a framework to carry out dynamic taint analysis in general, we rather tried to provide enough details for a precise algorithm that can be implemented by researchers based on their specific needs and their domain of study. In a context like security, researchers probably need to be able to do taint analysis on programs in an online manner where they can detect and respond to any suspicious activity, while for a binary analyst, an offline system that analyses an execution trace is more desirable.

[2]Taint sources, such as the `x86` instruction `rdtsc`, may produce tainted destination operands even if they have no tainted source operands. Such instructions are handled in Step 1 of the algorithm.

We have implemented a prototype tool to evaluate our proposed method for `x86` assembly language. Our implementation is an offline system: we collect an execution trace of the program we want to analyze and then perform further analysis on this trace. As shown in Algorithm 1, we start by identifying taint sources. This involves finding registers and memory sets that are passed to the program by system calls of interest. We used the `Udis86` disassembly library [25] for disassembly of `x86` instructions, but all the semantics of the instructions were handled by our code.

Our implementation uses bit-level granularity to propagate taint for code/data but it uses two sets of mapping functions and a mixed set of taint markings. For the computations involving the control flags register, we use taint markings that distinguish between taint sources at bit-level, i.e., distinct taint markings for each control flag bit. In order to propagate these symbolic taint marks we assigned a unique integer to each flag bit and so for every variable we keep a vector of integers to store the taint marks, one integer for each bit. Likewise a mapping function to handle flags data applies the semantics of the underlying operation to the integer vector of the variable. For any other computation in the program our implementation uses the standard notion of $\boldsymbol{T}$ and $\boldsymbol{F}$: $\boldsymbol{T}$ to denote tainted data and $\boldsymbol{F}$ otherwise. One bit of taint for every bit in the system is needed to do this so we used a bit-vector to propagate the taint for registers and memory locations.

This approach allows us to track every single control bit individually. This is particularly effective against emulation-based obfuscation where the virtual machine that interprets the protected program, usually use their own implementation of conditional control transfers. In `x86` assembly, conditional transfers are usually done immediately after an instruction that affects the `PSW flags` register like `cmp` or `test`. In a virtual machine however, the machine first saves the CPU flags register, picks the single bit that is going to be used for control transfer, e.g. `zero flag` or `overflow flag` and does the control transfer based on the actual value of that particular flag. Sometimes the bit is even used to index a jump table. In either case we need to be able to track every bit to determine whether a control transfer depends on some tainted input or not. This can be done using two sets of mapping functions: one to propagate taint among registers and memory addresses that use binary markings with bit-level granularity, and the other to propagate the markings used for control flags with different markings and again at bit-level granularity. As discussed in section IV, this approach gives good performance while achieving precise results in the analysis of heavily obfuscated code.

IV. Evaluation

One challenge in evaluating a taint analysis algorithm is showing that the analysis is *precise*, i.e., that a statement `s` is influenced by a taint source if and only if it is marked tainted by the algorithm. There is not a great deal of work on semantics-based formalization of taint analysis; Schwartz *et al.* define taint analysis in terms of functional semantics of an intermediate language [21]. One way to address this issue empirically is to execute the code exhaustively on an emulator, with different inputs, and monitor the execution. If a statement is marked tainted by the algorithm, then it should be input dependent and therefore produce at least two different behaviors for distinct inputs. However this approach has its own limitations. For instance we do not know how many different inputs should be examined for a statement to be affected in a program, and in general it may not be possible to execute a program on all possible inputs.

For ordinary compiler generated code (i.e., no obfuscation), our algorithm gives the same result as the standard dynamic taint analysis. Using the operational semantics of the language, which turns to different mapping functions, and distinct taint markings, we are able to *sanitize* the taint and avoid unnecessary taint spread and so prevent any taint explosion.

In order to show the effectiveness of our analysis against obfuscations, we have presented the results of two experiments. In the first experiment we measure the amount of code marked tainted using different taint approaches and the results are compared. Secondly we show how precise each taint approach is when it is used to recover the original logic of a *virtualized* program. Reverse engineering of the programs protected with emulation-based tools are known to be hard because in addition of adding heavy obfuscation to the programs, they emulate the underlying logic to make it more stealthy from the analyzer, i.e., they run the programs through an arbitrary virtual machine built specifically for that particular program. We evaluated our algorithm with obfuscated samples for two reasons: first, to show that our prototype implementation of the proposed taint analysis algorithm is able to successfully handle arbitrarily complex obfuscations, even emulation-based obfuscations; and second, to compare our approach with other existing approaches on obfuscated code. Moreover, the output of our taint analysis algorithm and existing ones should be the same for non-obfuscated programs (we have verified this with non-obfuscated versions of our test input programs).

We used six programs for our evaluation: four synthetic benchmarks, *binary-search*, *bubble-sort*, *huffman*, and *matrix-multiply*; and two malicious programs: *hunatcha*, a file dropper whose C source code was obtained from the VX Heavens web site [27], and *stuxnet*, the encryption routine taken from the decompiled code for the Stuxnet worm [11]. Each program was obfuscated using four different commercial obfuscators: *Code Virtualizer* [17] , *EXECryptor* [24], *Themida* [18] and *VMProtect* [26]. These programs read some data from input and do simple (*bin-search*) to moderately complex (*stuxnet*) computations on the input data. While the original (unobfuscated) programs are relatively simple, the obfuscated versions are significantly larger and have much more complex data and control flow characteristics (e.g., see Figure 5). We collected execution traces of the original and obfuscated binaries using a modified version of Ether [5].

For our experiments we used three different taint analysis approaches: *Byte-level*, *Bit-level* and *Enhanced*. The first two are standard taint analyses at byte-level and bit-level granular-

	Program	% Tainted Instructions		
		standard	*Bit-level*	*Enhanced*
EC	*binary-search*	30	23	19
	bubble-sort	30	26	18
	huffman	32	24	21
	hunatcha	30	30	27
	matrix-multiply	30	26	19
	stuxnet	31	28	17
VM	*binary-search*	48	48	7
	bubble-sort	69	67	7
	huffman	58	57	25
	hunatcha	45	45	17
	matrix-multiply	60	57	6
	stuxnet	64	61	7

TABLE I: Quantitative data for various taint analysis approaches

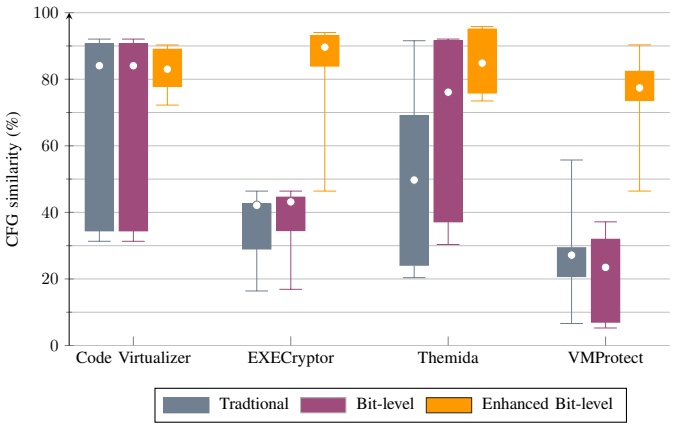


Fig. 4: Comparing different taint analysis algorithms

ity respectively; the third (Enhanced) is the approach discussed in this paper. Since we are interested in the flow of input values in each program through the trace, the initial taint markings are the same for all three algoithms, namely, the outputs of all system calls made by the program.

Table I shows the percent of the tainted instructions for each program/method. The rows *EC* and *VM* stand for programs protected using EXECryptor and VMProtect respectively. It can be seen from the table that in all cases, our algorithm marks a much smaller portion of the trace as tainted. As expected, running the standard taint analysis with bit-level granularity will also mark fewer instructions as tainted so it will produce a more accurate result than the byte- or word-level analysis but as the numbers show, it does not significantly improve the standard byte-level algorithm. This shows that doing dynamic taint analysis even at bit-level, is not effective when the code is obfuscated. Intuitively, the percentage of the trace being tainted largely depends on the program logic but it also depends on the obfuscation techniques and tools.

Our second experiment studies the effectiveness of different taint analysis approaches in simplifying the obfuscated versions of our benchmarks. The idea here is that the semantics of a program can be understood as a mapping from input values to output values and so deobfuscation becomes a problem of identifying and simplifying the code that effects this mapping [29]. Dynamic taint analysis is used to track the flow of values from inputs to outputs (we also need to handle implicit flows due to tainted control transfers; these details are orthogonal to the topic of this paper and so are omitted). Instructions involved in this input-to-output flow of data are then simplified using semantics-preserving transformations, while instructions not involved in this flow are eliminated. We then construct the control flow graph (CFG) of the *simplified* program and compare it to the CFG of the original (unprotected) program. The similarity of the original and simplified CFGs is crucially dependent on the precision of the taint analysis.

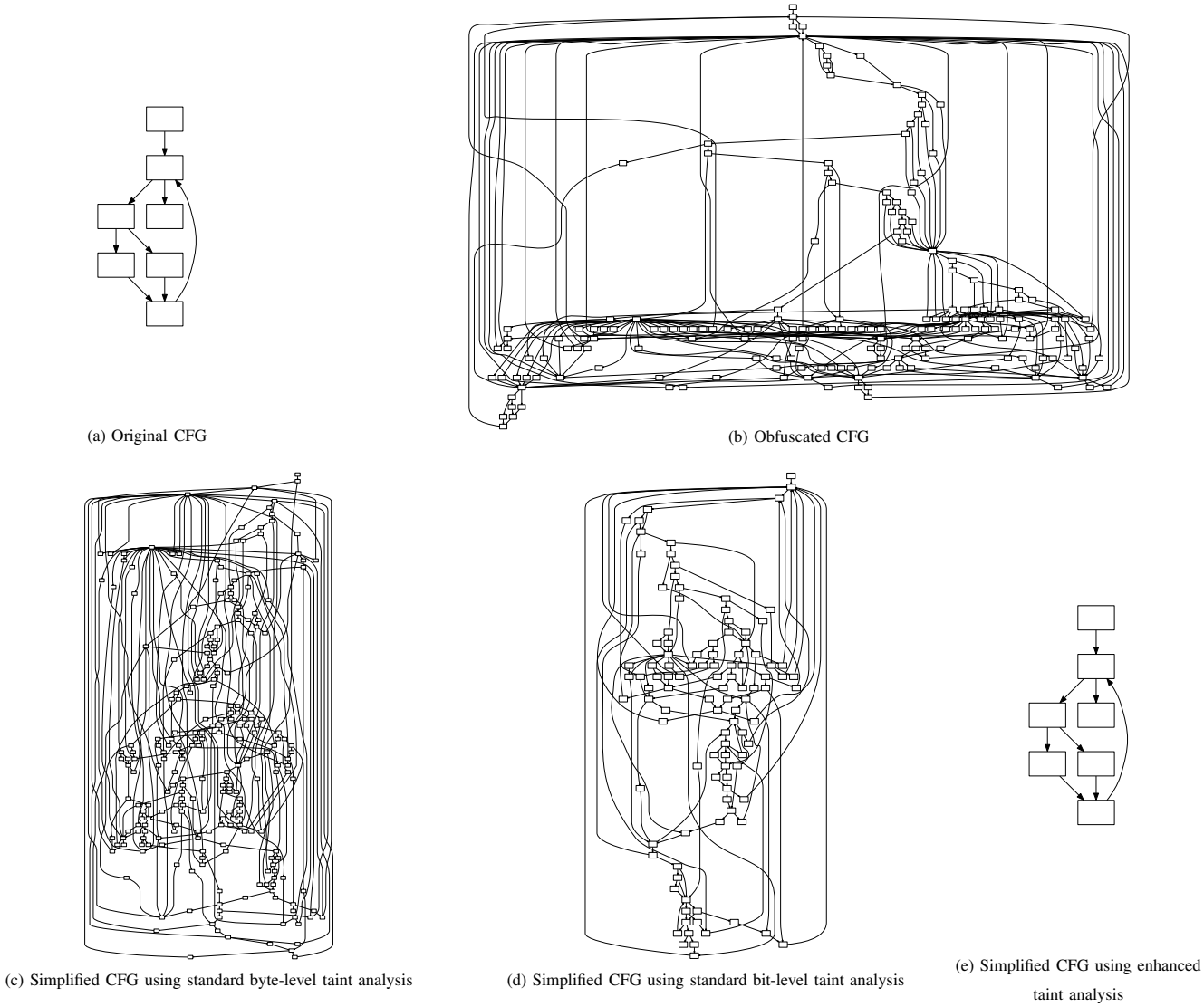
To measure CFG simlarity, we used a normalized version of a graph edit distance algoritm by Hu, Chiueh, and Shin [8]. This algorithm uses maximum bipartite matching to compute a correspondence between the vertices of the two CFGs G_1 and G_2 being compared, then uses this correspondence to determine the edit distance $\delta(G_1, G_2)$, i.e., the number of vertex and edge insertion/deletion operations necessary to tranform one graph to the other. To facilitate comprisons between CFGs of different sizes, we compute their *similarity* $sim(G_1, G_2)$ as the normalized edit distance:

$$sim(G_1, G_2) = 1 - \frac{\delta(G_1, G_2)}{|G_1| + |G_2|}$$

where $|G|$ is the size of the graph G and is given by the total number of vertices and edges in G. The computed similarity number ranges between 0 and 1 where a similarity of 1 means the graphs are identical.

Figure 4 shows the CFG similarity numbers of the original and simplified CFGs for the three different approaches to taint analysis. It can be seen that simplification using the enhanced bit-level taint analysis achieves the highest CFG similarity compared to standard byte- or bit-level taint analysis. While the results suggest that bit-level taint analysis is slightly better than byte-level analysis, it is still too imprecise to identify the data flow of the original program correctly. For programs protected with Code Virtualizer, there is not much difference between byte- and bit-level analysis meaning that both of the analysis produce the same result. This behavior entirely depends on the internal complexity of the obfuscators.

Figure 5 shows the control flow graphs for the binary-search program obfuscated using EXECryptor. Figure 5(a) shows the original CFG while (b) is the obfuscated programs. Figures 5(c), (d) and (e) show the simplified graphs with standard, bit-level and enhanced taint analysis algorithms respectively. It can be seen from the simplified CFGs that neither of the graphs in Figure 5(c) and (d) are similar to the original CFG. The reason is that both standard byte-level and bit-level taint analysis algorithms, while the bit-level analysis doing slightly better than the standard byte-level algorithm, are too imprecise in propagating the taint and over-taint irrelevant code making the analysis not not being able to resemble the original logic. Nevertheless, Figure 5(e) shows that with enhanced taint analysis we are able to recover the original logic of the obfuscated program suggesting that the enhanced algorithm is able to propagate the taint precisely.



(a) Original CFG

(b) Obfuscated CFG

(c) Simplified CFG using standard byte-level taint analysis

(d) Simplified CFG using standard bit-level taint analysis

(e) Simplified CFG using enhanced taint analysis

Fig. 5: Deobfuscation result of binary-search program protected by EXECryptor using different taint analysis methods

	Program	Time (seconds)		
		standard	*Bit-level*	*Enhanced*
EC	*stuxnet*	27.71	53.81	36.87
EC	*huffman*	21.21	21.24	26.68
VM	*stuxnet*	28.19	109.33	54.96
VM	*huffman*	196.03	518.11	449.73
Average		68.28	175.6	142.6

TABLE II: Quantitative data for various taint analysis approaches

The analysis was carried out on a machine with 2× quad-core 2.66 GHz Intel Xeon processors with 96 GB of RAM running Ubuntu Linux 12.04. The analysis speed for our four largest traces, *stuxnet* and *huffman* programs protected with EXECryptor and VMProtect is given in Table II. Table II shows the amount of time needed to run each of the three taint analysis algorithms on selected traces. The rows *EC* and *VM* stand for EXECryptor and VMProtect. The trace size for stuxnet and huffman protected by EXECryptor was nearly 5.4 and 6.8 million instructions long and for those protected by VMProtect was about 12.5 and 32.3 million instructions long. For our largest trace the speed of the analysis translates to nearly 164k instructions/second for byte-level, 61k instructions/seconds for bit-level and 72k instructions/seconds for enhanced taint analysis.

Increasing the number of distinct taint markings affects the amount of memory the algorithm needs but does not significantly affect analysis speed since once a bit gets its taint marks the propagation part is the same as before.

V. Related Work

Several researchers have discussed dynamic taint analysis techniques in recent years. Most of these works are applications of taint analysis; we are not aware of much work focusing on improving the accuracy of the analysis itself. The work conceptually closest to ours is that of Clause *et al* [3], which proposes a generic framework for dynamic taint

analysis. However, this paper does not discuss the precision of the algorithm nor its effectiveness against various obfuscation techniques. Schwartz *et al.* define dynamic taint analysis based on the operational semantics of the language [21]. However, they do not consider notions of distinct taint markings and different mapping functions. Cavallarro *et al.* [2] and Sarwar *et al.* [20] discuss approaches for defeating taint analyses.

Drewry et al. describe a bit-precise taint analysis system named *flayer* [7]. *Taintgrind* [28] is another taint analysis tool which is based on *flayer*. Despite carrying out the analysis at bit-level, these tools use the same notion of taint propagation as standard taint analysis, so neither is precise enough to deal with obfuscations or abnormal data-flow.[3]

VI. Conclusion

This paper describes the limitations of the standard taint analysis algorithm in dealing with obfuscation and tries to address these limitations. The enhanced taint analysis approach discussed in this paper extends the notion of taint annotation by using distinguished taint mark for each taint source. We also use functional semantics of the language as a means to precisely propagate the taint in a statement. We have considered the granularity in which the analysis is done and studied the precision and performance of each method. Our experiments show that the proposed approach is able to propagate the taint very precisely in codes obfuscated using sophisticated tools where the standard taint analysis results in over-tainting.

Our approach only considers a single execution of the program. To generalize the analysis, as a future work, one approach is to collect multiple traces of the program with different inputs and then run the taint analysis on the union of the collected traces.

Acknowledgments

This research was supported in part by the Air Force Office of Scientific Research (AFOSR) under grant no. FA9550-11-1-0191 and the National Science Foundation (NSF) under grants CNS-1115829, III-1318343, and CNS-1318955. The opinions, findings, and conclusions expressed in this paper are solely those of the authors and do not necessarily reflect the views of AFOSR or NSF.

References

[1] G. Arboit. A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, pages 102–110, 2002.

[2] L. Cavallaro, P. Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Stony Brook University, Stony Brook, New York*, 2007.

[3] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[4] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 IEEE International Conference on Computer Languages*, pages 28–38.

[5] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 51–62, Oct. 2008.

[3]We were not able to compile the code for *flayer*, but have verified our hypothesis with *taintgrind* which is based on *flayer* and is as precise as *flayer* in terms of granularity.

[6] S. Drape et al. Intellectual property protection using obfuscation. *Proceedings of SAS 2009*, 4779:133–144, 2009.

[7] W. Drewry and T. Ormandy. Flayer: Exposing application internals. *WOOT*, 7:1–9, 2007.

[8] X. Hu, T.-C. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proc. ACM Conference on Computer and Communications Security*, pages 611–620, Nov. 2009.

[9] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[10] J. Kong, C. C. Zou, and H. Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24. ACM, 2006.

[11] Laurelai. Partial stuxnet source decompiled with hexrays. `https://github.com/Laurelai/decompile-dump/blob/master/output/016169EBEBF1CEC2AAD6C7F0D0EE9026/016169EBEBF1CEC2AAD6C7F0D0EE9026.c`, 2010.

[12] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 198–209. IEEE, 2004.

[13] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[14] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[15] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. *Automatically hardening web applications using precise tainting*. Springer, 2005.

[17] Oreans Technologies. Code virtualizer: Total obfuscation against reverse engineering. `http://www.oreans.com/codevirtualizer.php`.

[18] Oreans Technologies. Themida: Advanced windows software protection system. `http://www.oreans.com/themida.php`.

[19] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2006.

[20] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *10th International Conference on Security and Cryptography (SECRYPT)*, 2013.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[22] A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74. ACM, 2009.

[23] B. Spasojević. Code deobfuscation by optimization. `http://optimice.googlecode.com/files/Deobfuscation-27C3.pdf`.

[24] StrongBit Technology. EXECryptor – bulletproof software protection. `http://www.strongbit.com/execryptor.asp`.

[25] V. Thampi. Udis86: Disassembler Library for x86 and x86-64. `http://udis86.sourceforge.net/`.

[26] VMProtect Software. VMProtect – New-generation software protection. `http://www.vmprotect.ru/`.

[27] VX Heavens. Vx heavens, 2011. http://vx.netlux.org/.

[28] Wei Ming Khoo. Taintgrind: a Valgrind taint analysis tool. `https://github.com/wmkhoo/taintgrind`.

[29] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. Technical report, Department of Computer Science, The University of Arizona, May 2014.

[30] H. Yin and D. Song. *Automatic Malware Analysis: An Emulator Based Approach*. Springer, 2012.

[31] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.