DIMSUM: Discovering Semantic Data of Interest from Un-mappable Memory with Confidence

Zhiqiang Lin[†], Junghwan Rhee[‡], Chao Wu[§], Xiangyu Zhang[§], Dongyan Xu[§]

[†]Department of Computer Science
University of Texas at Dallas, Richardson, TX
zhiqiang.lin@utdallas.edu

[‡]NEC Laboratories America
Princeton, NJ
rhee@nec-labs.com

[§]CERIAS and Department of Computer Science
Purdue University, West Lafayette, IN
{wu233,xyzhang,dxu}@cs.purdue.edu

Abstract

Uncovering semantic data of interest in memory pages without memory mapping information is an important capability in computer forensics. Existing memory mapping-guided techniques do not work in that scenario as pointers in the un-mappable memory cannot be resolved and navigated. To address this problem, we present a probabilistic inference-*based approach called DIMSUM to enable the recognition of data structure instances from un-mappable memory. Given a set of memory pages and the specification of a target data structure, DIMSUM will identify instances of the data structure in those pages with quantifiable confidence. More specifically, it builds graphical models based on boolean constraints generated from the data structure and the memory page contents. Probabilistic inference is performed on the graphical models to generate results ranked with probabilities. Our experiments with real-world applications on both Linux and Android platforms show that DIMSUM achieves higher effectiveness than non-probabilistic approaches without memory mapping information.*

1 Introduction

A common task in computer forensics is to uncover semantic information of interest, such as passwords, contact lists, chat content, cookies, and browsing history from raw memory. A number of recent efforts have demonstrated the capability of uncovering instances of data structures defined in a program. These efforts fall into two main categories: those relying on *memory mapping information* and those based on *value invariant patterns*. Existing solutions such as KOP [6], REWARDS [17] and SigGraph [16] belong to the first category, all working by traversing pointers between data structures to identify the instances of interest. As such, they require pointers to be resolvable (and thus trackable) in the memory image. KOP and REWARDS further require that each target data structure instance be reachable (via pointers) from global or stack variables.

Unfortunately, such memory mapping information is not always available. Yet it is desirable for a cyber crime investigator to have the capability of uncovering meaningful forensics information from a set of memory pages *without* memory mapping information. One such forensics scenario is as follows: Imagine a cyber crime suspect runs and then terminates an application (e.g., a web browser). He/she even cleans up the privacy/history data in the disk in order not to leave any evidence. At that moment, however, some of the memory pages previously belonging to the terminated application process may still exist for a non-trivial period of time – with intact content but without the corresponding page table or system symbol table (to be explained in greater detail in Section 2). While these "dead" memory pages may contain data of forensic interest, existing memory mapping-based forensics techniques (e.g., [6, 16, 17]) will not be able to uncover them. This is because, without memory mapping information, they will not be able to resolve and navigate through pointers in the dead pages. Similar observations have also been reported in [27, 28], which advocate the need for techniques to recover data from memory pages marked as free by the operating system (OS).

In addition to the above scenario of "dead pages left by a terminated process", there are other computer forensics scenarios that require analyzing partial memory image without memory mapping information. For example, after a sudden power-off, a subset of the memory pages belonging to a running process may still exist in the disk due to page swapping. But the memory mapping information maintained by the OS kernel for that process is lost. As another forensics scenario, a cyber crime investigator may only possess a subset of the physical memory pages for investigation, due to the physical damage or tampering to the subject computer.

To analyze partial, un-mappable memory image, another category of existing approaches (e.g., [4, 11, 24, 25]) leverage value-invariant signatures of data structures (e.g., "data structure field x having a special value or value range"). These techniques are effective if unique signatures can be generated for the subject data structures. However, such a signature may not exist for a data structure, as illustrated in [6, 16].

Motivated by the need in forensics and the limitations of existing solutions, we develop a new approach called DIMSUM[1], which is capable of uncovering data structure instances of forensics interest from a set of physical memory pages without memory mapping information. Moreover, DIMSUM does not require the presence of unique value invariant patterns and will remain effective even with an incomplete subset of memory pages of an application process. Such capability is useful not only in memory forensics, but also in the more generic settings of memory data analysis.

DIMSUM is based on *probabilistic inference*, which is widely used in computer vision, specification extraction [14, 18, 3], and software debugging [14, 18, 20, 23, 30]. Given a set of memory pages and the definitions of data structures of interest, DIMSUM is able to identify instances of the data structures in those pages. More specifically, by leveraging a probabilistic inference engine, DIMSUM automatically builds graphical models from the data structure specification and input page contents, and translates them into *factor graphs* [30], on which probabilistic inference will be performed to extract target data structure instances quantified with probabilities. The graphical models probabilistically encode both the primitive value fields and the point-to relations between data structures to tolerate the uncertainty due to lack of field type and memory mapping information. We point out that the purpose of DIMSUM is different from that of another statistical technique Laika [9]: DIMSUM uncovers data structure *instances* from binary memory pages; whereas Laika infers data structure type *definitions* from binary programs. As such, their entailed assumptions and modeling techniques are completely different.

The salient features of DIMSUM are as follows: (1) It recognizes data structure instances of interest with high confidence. Compared with brute force pattern matching methods, it consistently achieves lower false positive rate. (2) It is robust in highly challenging memory forensics scenarios, where there is no memory mapping information and only an incomplete subset of memory pages are available. We have evaluated DIMSUM using a number of real-world applications on both Linux and Android platforms, and demonstrated the effectiveness of DIMSUM.

2 Background and System Overview

2.1 Observations

DIMSUM was firstly motivated by the "dead memory pages left by terminated processes" scenario as described in Section 1. More specifically, we notice that, when a process is terminated, neither Windows nor Linux OS clears the content of its memory pages. We believe one of the reasons is to avoid memory cleansing overhead. Moreover, Chow et al [8] found that many applications let sensitive data stay in memory after usage instead of "shredding" them. Even if an application performs data "shredding", it is still possible that a crash happens before the shredding operation, leaving some sensitive data in the dead memory pages.

Secondly, we also observe that dead pages may remain intact for a non-trivial duration, which we call their *death-span*. In fact, we observe that the death-span of dead pages of a Firefox process can last up to 50 minutes after the process terminates, in a machine with 512 MB RAM. If the machine has a larger RAM or the workload after Firefox's termination is not as memory-intensive, the death-span of dead pages may be even longer. A similar study on the age of freed user process data on Windows XP (SP2) [27] has shown that large segments of pages can survive for nearly 5 minutes in a lightly loaded system, and smaller segments and single pages may be found intact for up to 2 hours.

Finally, we observe that, for a terminated process, the corresponding memory mapping information maintained by the OS kernel, such as the process control block and page table, are likely to disappear (i.e., be reused) much sooner. The much shorter dead-span of kernel objects (typically a few seconds) – contrary to that of dead application pages – is due to the fact that kernel objects are maintained as slab objects by the kernel [5], which uses LIFO as the memory recycling policy; whereas memory pages of processes are managed by the buddy system [5] which groups memory frames into lists of blocks having 2^k contiguous frames, and hence page frames tends to have longer dead-span.

[1] DIMSUM stands for "Discovering InforMation with Semantics from Un-mappable Memory."

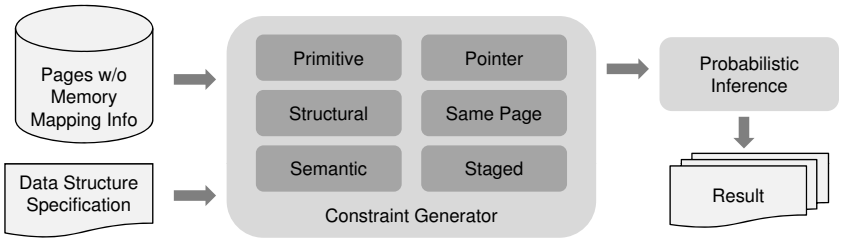


Figure 1. Overview of DIMSUM.

2.2 Challenges

Our observations above motivate the development of DIMSUM. Compared with existing approaches, DIMSUM poses a number of new challenges. The first challenge is the absence of memory mapping information. Consequently, given a set of memory pages, there is little hint on which pages belong to which process, let alone the sequencing of physical pages in the virtual address space of a process. Even if we can identify some pointers in a page, we still cannot follow those pointers without the address mapping information.

The second challenge is that DIMSUM may accept an incomplete subset of memory pages of a process as input. In this case the application data that reside in the absent pages cannot be recovered. However, such data could be useful for the recognition of application data that reside in the input pages, especially when a pointer-based memory forensics technique is employed.

The third challenge is the absence of type/symbolic information for dead memory. To map the raw bits and bytes of a memory page to meaningful data structure instances, type information is necessary. For example, if the content at a memory location is 0, its type could be integer, floating point, or even pointer. If these bits and bytes belong to the live memory, symbolic information is available and they can be typed through reference path (as in [6]). To DIMSUM, however, such information is not available.

2.3 Overview of DIMSUM

To address the above challenges, we take a *probabilistic inference* and *constraint solving* approach. Fig. 1 shows the key components and operations of DIMSUM. The input of the system includes: (1) a subset of memory pages from a computer and (2) the specifications of data structure(s) of interest. Note that a data structure specification includes field offset and type information, which can be obtained from either application documentation, debugging information, or reverse engineering [17, 15, 26].

A key component of DIMSUM, *constraint generator*, transforms the data structure specification into constraint templates that are instantiated by the input memory pages. These templates describe correlations dictated by data structure field layout, and include *primitive*, *pointer*, *structural*, *same-page*, *semantic*, and *staged* constraints.

Next, the *probabilistic inference* component automatically transforms all the constraints into a factor graph [30], and efficiently computes the marginal probabilities of all the candidate memory locations for the data structure of interest. Finally, it outputs the result based on the probability rankings.

3 DIMSUM Design

The essence of DIMSUM is to formulate the data structure recognition problem as a probabilistic constraint solving problem. We first use a working example to demonstrate the basic idea, which relies on solving boolean constraints.

3.1 Working Example

```
struct utmplist {
    00: short int ut_type;
    04: pid_t ut_pid;
    08: char ut_line[32];
    40: char ut_id[4];
    44: char ut_user[32];
    76: char ut_host[256];
   332: long int ut_etermination;
   336: long int ut_session;
   340: struct timeval ut_tv;
   348: int32_t ut_addr_v6[4];
   364: char __unused[20];
   384: struct utmplist *next;
   388: struct utmplist *prev;
}
```

Figure 2. Data structure definition of our working example.

Ideally, our technique will take (1) the data structure specification such as the one defined in Fig. 2, which is the `utmplist` data structure showing a list of last logged users in a Linux utility program `last` and (2) a set of memory pages, and then try to identify instances of the data structure in the pages. The idea is to first generate a set

of constraints from the given data structure. For example, given the predicate definitions presented in Table 1 and assuming a 32-bit machine, the generated constraint for the `utmplist` structure would be:

$$\begin{aligned}\texttt{utmplist}(a) \quad &\rightarrow \mathcal{I}_{\texttt{ut\_type}}(a) \wedge \mathcal{I}_{\texttt{ut\_pid}}(a+4) \wedge \\ &\mathcal{C}_{\texttt{ut\_line}}(a+8)[32] \wedge \mathcal{C}_{\texttt{ut\_id}}(a+40)[4] \wedge \\ &\mathcal{C}_{\texttt{ut\_user}}(a+44)[32] \wedge \mathcal{C}_{\texttt{ut\_host}}(a+76)[256] \wedge \\ &\mathcal{I}_{\texttt{ut\_session}}(a+336) \wedge \mathcal{I}_{\texttt{ut\_etermination}}(a+332) \wedge \\ &\mathcal{I}_{\texttt{ut\_tv.tv\_sec}}((a+340)) \wedge \mathcal{I}_{\texttt{ut\_tv.tv\_usec}}((a+344)) \wedge \\ &\mathcal{I}_{\texttt{ut\_addr\_v6}}((a+348)[4]) \wedge \mathcal{C}_{\texttt{\_unused}}((a+364)[20]) \wedge \\ &\mathcal{P}_{\texttt{next}}(a+384) \wedge \texttt{utmplist}(*(a+384)) \wedge \\ &\mathcal{P}_{\texttt{prev}}(a+388) \wedge \texttt{utmplist}(*(a+388)) \wedge \\ &*(a+4)_{ut\_pid} \geq 0 \end{aligned} \tag{1}$$

Note that the subscripts are used to denote field names. Intuitively, the above formula means that, if the location starting at a sees an instance of `utmplist`, location a will contain an integer, location $a+4$ will contain another integer, $a+8$ will contain a character array with size 32, and so on. The constraint also dictates that the locations pointed to by pointers at $a+384$ and $a+388$ contain instances of `utmplist` as well. These are called *structural constraints* as they are derived from the type structure. We may also have *semantic constraints* that predicate on the range of the value at an address. The term at the end of the constraint specifies that field `ut_pid` should have a non-negative value. Semantic constraints can be provided by the user based on domain knowledge.

Besides the above constraints, we also extract a set of *primitive constraints* by scanning the pages. These constraints specify what primitive type each location has. We consider seven primitive types: *int*, *float*, *double*, *char*, *string*, *pointer* and *time*. Here, we leverage the observation that deciding if a location is an instance of a primitive type, such as a pointer, can often be achieved by looking at the value. Suppose that addresses 0, 4, 8, and 12 have been determined to contain an integer, an integer, a non-negative integer, and a char array with size 16, primitive constraints $\mathcal{I}(0), \mathcal{I}(4), \mathcal{I}(8), \mathcal{C}(12)[16]$ (defined in Table 1) will be generated. By conjoining the structural, semantic, and primitive constraints, we can use a solver to produce satisfying valuations for $utmplist(a)$, which essentially identifies instances of the given type. With the above constraints, $a=0$ is not an instance because $\mathcal{C}(a+8)[32]$ is not satisfied. In contrast, $a=4$ might be an instance.

3.2 Probabilistic Inference

However, the example in Section 3.1 faces a number of real-world issues in the context of DIMSUM:

Uncertainty in primitive constraints: While values of primitive types have certain attributes, it is in general hard to make a binary decision on a type predicate by looking at the value. In such cases, we expect that our solution is able to reason with probabilities.

Predicate	Definitions
$\tau(x)$	The location at x is an instance of a user-defined type τ
$\mathcal{I}(x)$	The location at x is an integer.
$\mathcal{F}(x)$	The location at x is a floating point value.
$\mathcal{D}(x)$	The location at x is a double floating point value.
$\mathcal{S}(x)$	The location at x is a string.
$\mathcal{C}(x)$	The location at x is a char.
$\mathcal{P}(x)$	The location at x is a pointer.
$T(x)[y]$	The location at x is an array of size y, with each element of type T.

Table 1. Predicate definitions (used throughout the paper)

Absence of page mappings: As discussed in Section 2, a pointer value is essentially a *virtual* address. Without memory mapping information, for constraints like $\mathcal{S}(*a)$, we cannot identify the page being pointed to by a and thus cannot decide if a points to a string.

Incompleteness: We may see only part of a data structure, e.g., some elements in a linked list may be missing. Our solution should be able to resolve constraints for such cases.

To address the above issues, we formulate our problem as a probabilistic inference problem [23, 30]. Initial probabilities are associated with individual constraints, representing the user's view of uncertainty. The probabilities are efficiently propagated, aggregated, and updated over a graphical representation called *factor graph* [30]. After convergence, the final probabilities of boolean variables of interest can hence be queried from the factor graph. We next elaborate via an example.

We simplify the case in the Fig. 2 by considering only the pointer fields, i.e., fields at offsets 384 and 388. For a given address a, let boolean variable x_1, x_2, and x_3 denote $T_{utmplist}(a)$, $P_{next}(a+384)$, and $P_{prev}(a+388)$, respectively. The structural constraint is simplified as follows.

$$x_1 \rightarrow x_2 \wedge x_3 \tag{2}$$

Assume the structural pattern is unique across the entire system, meaning that there are no other data structures across the system with the same structural pattern. In particular for the above pattern, if we observe two consecutive pointers in memory, we can be assured that they must be part of an instance of `struct utmplist`, we have the following constraint.

$$x_1 \leftarrow x_2 \wedge x_3 \tag{3}$$

With this constraint, when we observe $x_2=1$ and $x_3=1$, we can infer $x_1=1$, meaning that there is an instance of `struct utmplist` at address a. If $x_2=1$ and $x_3=0$, we infer that $x_1=0$.

In general, assume there are m constraints C_1, C_2..., and C_m on n boolean variables x_1, x_2, ..., and x_n. Functions

f_{C_1} f_{C_2}, ..., and f_{C_n} describe the valuation of the constraints. For instance, let C_1 be Equation (2), $f_{C_1}(x_1 = 1, x_2 = 1, x_3 = 0) = 0$. Since all the constraints need to be satisfied, the function representing the conjunction of the constraints is hence the product of the individual constraint functions, as shown in Equation (4).

$$f(x_1, x_2, ..., x_n) = f_{C_1} \times f_{C_2} \times ... \times f_{C_m} \quad (4)$$

In DIMSUM, we often cannot assign a boolean value to a variable or a constraint. Instead, we can make an observation about the likelihood of a variable being true. For instance, from the value stored at offset $a+384$, we can only say that it is likely a pointer. Moreover, if the structural pattern of $T_{utmplist}$ is not unique, i.e., other data structures may also have such a pattern, we can similarly assign a probability to constraint (3) according to the number of data structures sharing the same pattern.

Assume we use a set of boolean variables x_1, x_2, ..., x_n to represent type predicates. Probabilities are associated with variables and constraints. In our previous example, assume that we are 100% sure that $x_1 \rightarrow x_2 \wedge x_3$ (C_1); 80% sure that $x_1 \leftarrow x_2 \wedge x_3$ (C_3) because other data structures manifest a similar structural pattern; 90% sure that x_2 is a pointer (C_2); 90% sure that x_3 is a pointer (C_4). We have probabilistic functions:

$$f_{C_1}(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } (x_1 \rightarrow x_2 \wedge x_3) = 1 \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

$$f_{C_2}(x_2) = \begin{cases} 0.9 & \text{if } x_2 = 1 \\ 0.1 & \text{otherwise} \end{cases} \quad (6)$$

$$f_{C_3}(x_1, x_2, x_3) = \begin{cases} 0.8 & \text{if } (x_1 \leftarrow x_2 \wedge x_3) = 1 \\ 0.2 & \text{otherwise} \end{cases} \quad (7)$$

$$f_{C_4}(x_3) = \begin{cases} 0.9 & \text{if } x_3 = 1 \\ 0.1 & \text{otherwise} \end{cases} \quad (8)$$

With these probabilistic constraints, the joint probability function is defined as follows [23, 30].

$$p(x_1, x_2, ..., x_n) = \frac{f_{C_1} \times f_{C_2} \times ... \times f_{C_m}}{Z} \quad (9)$$

$$Z = \sum_{x_1,...,x_n} (f_{C_1} \times f_{C_2} \times ... \times f_{C_m}) \quad (10)$$

In particular, Z is the normalization factor [23, 30].

It is often more desirable to further compute the marginal probability $p_i(x_i)$ as follows.

$$p_i(x_i) = \sum_{x_1} \sum_{x_2} ... \sum_{x_{i-1}} \sum_{x_{i+1}} ... \sum_{x_n} p(x_1, x_2, ..., x_n) \quad (11)$$

In other words, the marginal probability is the sum over all variables other than x_i. Variable x_i often predicates on a given address having the type we are interested in. Hence, in order to discover the instances of the specific type, DIMSUM orders memory addresses by their marginal probabilities.

x_1	x_2	x_3	$f_{C_1}(x_1, x_2, x_3)$	$f_{C_2}(x_2)$	$f_{C_3}(x_1, x_2, x_3)$	$f_{C_4}(x_3)$
0	0	0	1	0.1	0.8	0.1
0	0	1	1	0.1	0.8	0.9
0	1	0	1	0.9	0.8	0.1
0	1	1	1	0.9	0.2	0.9
1	0	0	0	0.1	0.8	0.1
1	0	1	0	0.1	0.8	0.9
1	1	0	0	0.9	0.8	0.1
1	1	1	1	0.9	0.8	0.9

Table 2. Boolean constraints with probabilities.

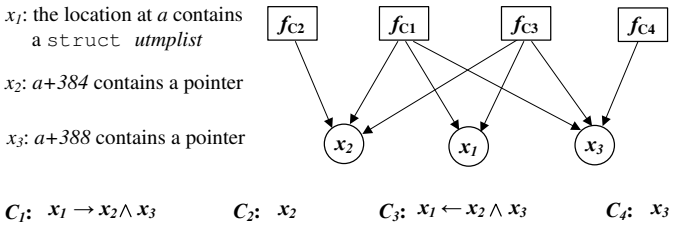


Figure 3. Factor graph example.

Consider the previous example. Table 2 presents the values of the four probability constraint functions for all possible variable valuations.

$$\begin{aligned} p(x_1 = 1) &= \frac{\sum_{x_2,x_3} f_{C_1}(1, x_2, x_3) \times f_{C_2}(x_2)}{\sum_{x_1,x_2,x_3} f_{C_1}(x_1, x_2, x_3) \times f_{C_2}(x_2)} \\ &= \frac{0 \times 0.1 + 0 \times 0.1 + 0 \times 0.9 + 1 \times 0.9}{1 \times 0.1 + 1 \times 0.1 + ... + 1 \times 0.9} \\ &= \frac{0.9}{2.9} = 0.31 \end{aligned} \quad (12)$$

$$\begin{aligned} p(x_2 = 1) &= \frac{1 \times 0.9 + 1 \times 0.9 + 0 \times 0.9 + 1 \times 0.9}{2.9} \\ &= 0.93 \end{aligned} \quad (13)$$

Assume only constraints C_1 and C_2 are considered, Equation (12) describes the computation of the marginal probability of $p(x_1 = 1)$, i.e., the probability of the given address being an instance of `struct utmplist`. Equation (13) describes the marginal probability of $p(x_2 = 1)$. Note that it is different from the initial probability 0.9 in f_{C_2}. Intuitively, the value assigned in f_{C_2} is essentially an observation, which does not necessarily reflect the intrinsic probability. In other words, the initial probability in f_{C_2} is what we believe and it reflects only a local view of the constraint, whereas the computed probability represents

a global view with all initial probabilities over the entire system being considered.

Similarly, when all four constraints are considered, we can compute $p(x_1 = 1) = 0.71$. Intuitively, compared to considering only C_1 and C_2, now we also have high confidence on x_3 (C_4) and we have confidence that as long as we observe x_2 and x_3 being true, x_1 is very likely true (C_3). Such information raises the intrinsic probability of x_1 being true.

Note that depending on the number of variables and the number of constraints, the computation in Equation (11) could be very expensive because it has to enumerate the combinations of variable valuations. *Factor graph* [23, 30, 18] is a representation for probability function that allows highly efficient computation. In particular, a factor graph is a bipartite graph with two kinds of nodes. A *factor node* represents a factor in the function, e.g., f_{C_i} in Equation (9). A *variable node* represents a variable in the function, e.g., x_i in Equation (9). Edges are introduced from a factor to the variables of the factor function. Fig. 3 presents the factor graph for the probability function in the previous example. The *sum-product* algorithm [23, 30] can leverage factor graphs to compute marginal probabilities in a highly efficient way. The algorithm is iterative. In particular, probabilities are propagated between adjacent nodes through message passing. The probability of a node is updated by integrating the messages it receives. The algorithm terminates when the probabilities become stable. At a high level, one can consider initial probabilities as energy applied to a mesh such that the mesh transforms to strike a balance and minimize free energy. Probabilistic inference has a wide range of successful applications in artificial intelligence, information theory and debugging [18, 14]. In this paper, DIMSUM is built on a probabilistic inference framework called *Infer.NET* [20].

In DIMSUM, to conduct probabilistic reasoning using the factor graph (FG), we first assign a boolean variable to each type predicate, indicating if a specific address holds an instance of a given type. We create a variable for each type of interest for each memory location. In other words, if there are n data structures of interest and m memory locations, we would generate $n * m$ boolean variables. We will introduce a pre-processing phase that can reduce the number of such variables by reducing m. Then constraints are introduced. A constraint is essentially a boolean formula on the boolean variables. Initial probabilities are assigned to these constraints to express uncertainty. The constraints and initial probability assignments are programmed using scripts. FGs are then constructed by these scripts using *Infer.NET* engine. After that, data structure instances can be identified by querying the probabilities of the corresponding boolean variables. Those within the highest-probability cluster are reported to the user.

4 Generating Constraints

We now explain how to generate the constraints involved in the FGs for memory forensics. The constraints fall into the following categories: *primitive constraints* that associate initial probabilities to individual boolean variables; *structural constraints* that describe field structures; *pointer constraints* that describe dependencies between a data structure and those being pointed to by its pointer fields; *same-page constraints* dictating multiple data structures reside in the same physical page; *semantic constraints* that are derived from the semantics of the given data structures. All these constraints are associated with initial probabilities. They are conjoined and updated by the inference engine.

4.1 Primitive Constraints

Primitive constraints allow us to assign initial probabilities to boolean variables. Sample primitive constraints are f_{C_2} and f_{C_4} in Eq. (6) and (8) in Section 3. A primitive constraint is translated to a factor node in FG. It has only one outgoing edge to the boolean variable (Fig. 3). We consider the following primitive types: `int`, `float`, `double`, `char`, `string`, `pointer` and `time`.

Pointer: To decide the initial probability of a boolean variable denoting that a memory location has a pointer, we check whether the value of 4 contiguous bytes starting at a given location is within the virtual address space of a process (e.g., in the `.data`, `.bss`, `.heap`, and `.stack` sections). If true, we assign a HIGH initial probability (0.9) to the primitive pointer constraint, representing our belief that the given location is likely a pointer. The other primitive constraints for the same location would be assigned a LOW (0.1) initial probability. Note that setting HIGH/LOW initial probabilities is a standard practice in probabilistic inference. They do not reflect the intrinsic probabilities of the boolean variables but rather what we believe. The absolute values of initial probabilities are hence *not* meaningful. NULL pointers with value 0 could be confused with a character or an integer. We will discuss how to handle them later.

String: To decide the initial probability of a string (a `char` array), we inspect the bytes starting with the given location. Firstly, a string ends with a NULL byte. Secondly, a string often contains the printable ASCII ($[32, 126]$) or some special characters such as carriage return (CR), newline (LF), and tab (Tab). If the two conditions are satisfied, the string constraint is set to HIGH, and other primitive constraints are set to LOW. It is possible that the bytes starting at x look like both a string and an integer. A unique advantage of probabilistic inference is that we can assign HIGH probabilities to multiple primitive constraints on x. Intuitively, it means we believe it could be multiple types.

Assigning multiple HIGH probabilities regarding the same memory location allows the location to try different roles during inferencing and we do not need to make the decision upfront. The inference process will eventually make the decision, by considering the probabilities from other parts of the FG through their dependencies.

Char: If a field with a `char` type is packed with other fields, that is, it is not padded to the word boundary, it becomes hard to disambiguate a char value from a byte that is just part of an integer or a floating point value. We have to set the probability to HIGH for all these primitive constraints. Fortunately, a `char` field is often padded. Hence, we can limit our test to offsets aligned with the word boundary. More particularly, we only assign a HIGH probability to locations whose four bytes values fall into $[0, 255]$.

Int: Compared to the above primitive types, integers have less attributes to allow disambiguation. Theoretically, any four bytes could be a legitimate integer value. In some cases, we are able to leverage semantic constraints to avoid assigning HIGH probabilities. For instance, it is often possible to find out from the data structure specification that an integer timeout field must have the value in $[0, 2^{10}]$. We could use such semantic information to assign LOW probabilities to values outside that range.

Float/double: According to the standard of floating-point format representation defined in IEEE 754 [2], we know the numerical value n for a float variable is: $n = (1 - 2s) \times (1 + m \times 2^{-23}) \times 2^{e-127}$, where s is a sign bit (zero or one), m is the significand (i.e., the fraction part), and e is the exponent. Fig. 4 shows this representation.

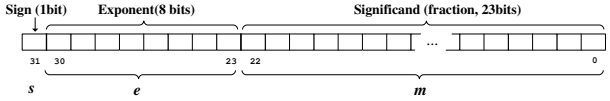


Figure 4. Float Point Representation.

Now if we examine the value of a floating point variable, suppose $s = 0$ and $e = 0$, then the numerical value is very small, and it is within $[0, 2^{-126}]$. Thus, we could infer that most floating point values have their leftmost 9 bits set with at least one bit. If all the leftmost 9 bits have been set with 1 (i.e., $s = 1$, $e = 255$), then the numerical value for such floating point variable is within $[-2^{128}, -2^{105}]$, which is a very large negative value. If the sign bit is 0 (i.e., $s = 0$, $e = 255$), then the numerical value is within $[2^{105}, 2^{128}]$, which is a very large positive value. In practice, we *believe* floating point values rarely fall into such ranges.

Therefore, we check the hexadecimal value at page offset x, i.e., $*x$, if $*x <$ `0x007fffff`, `0x7f800000` $< *x <$ `0x7f8fffff`, or `0xff800000` $< *x <$ `0xffffffff`, we set the initial probability of $\mathcal{F}(x)$ to LOW, otherwise HIGH. Double type is handled similarly. The details are elided.

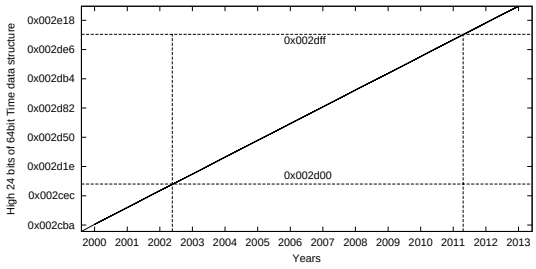


Figure 5. Common high bits in a time data structure.

Time: Time data structures are often part of many interesting data structures. A time data structure maintains the cumulative time units (e.g., seconds or microseconds) since a specific time in the past. Its bit representation has a general property that high bits are less frequently updated than lower bits. It allows us to create constraints to infer time data structures using common bit fields for all time values during a given period.

For example, Fig. 5 shows the values of the highest 24 bits of a time data structure of 64 bits over a period of time. During the period between mid-2002 and mid-2011, the highest 24 bits have the common value *0x002d*. These constraints can be used to infer time object instances.

Similarly, in 32-bit Unix systems, the time data structure has 32 bits. The four highest bits are updated around every 8.5 years.

Lastly, zeros present an interesting case because they could have multiple meanings: an integer with the value 0; an empty string; a null pointer; and so on. We assign HIGH probabilities to all these types except for cases in which the fields in the vicinity are also zeros. The reason is that consecutive zeros often imply unused memory regions. In particular, if the number of consecutive zeros exceed the size of the data structure we are interested in, the probability is set to LOW. In general, the probability is reversely proportional to the length of consecutive zeros.

4.2 Structural Constraints

As an input to DIMSUM, the data structure specification includes the field offsets and field types of the data structures. For instance, if a target data structure T has a pointer field of T_x type, T_x's definition is often transitively included as well. Then we translate each type into a boolean structural constraint describing the dependencies between the data structure and its fields. Eventually, the boolean constraints are modeled in the factor graph automatically.

A structural constraint is intended to reflect the dependence that, if a given location x is an instance of T, the corresponding offsets of x must be of the field types described in T's definition. An example of such constraint was introduced in Eq. (1) in Section 3.1. In particular, for each memory location, we introduce a boolean variable to predicate if it is an instance of T. We also introduce a factor node to represent the constraint. Edges are introduced between the factor node and the newly introduced variable and the variables describing the corresponding field types. A sample factor graph after such process is the subgraph rooted at f_{C_1} in Fig. 3. Since the constraint is always certain, meaning as long as x is of type T, its offsets must follow the structure dictated by T's definition. The probability of structural constraints is always 1.0, meaning that such constraints must hold (see Eq. (5) in Section 3).

4.3 Pointer Constraints

If a field $a+f$ is a pointer $T*$, in the structural constraint, besides forcing $a+f$ to be a pointer, we should also dictate $*(a+f)$ be of type T. In particular, we will add boolean variables $T(*(a+f))$ to the structural constraint. Note that T could be a primitive type, a user defined type, or a function pointer. Variables $\texttt{utmplist}(*(a+384))$ and $\texttt{utmplist}(*(a+388))$ in Equation (1) are examples. Ideally, these variables have been introduced at the time when we type the page of the pointer target (e.g., the page that $*(a+384)$ points to), we only need to introduce edges from the factor node to such variables.

However, since we do not have page mapping information, it is impossible to identify the physical location of the pointer target and the corresponding boolean variable. Fortunately, we observe that the lower 12 bits of a virtual address indicates the offset within a physical page. Hence, while we cannot locate the concrete physical page corresponding to the given address, we can look through all physical pages and determine if there are some pages that have the intended type at the same specified offset.

From now on, we denote a memory location with symbol a^p, with a being the page offset and p the physical page ID. Hence, a boolean variable predicating a location a^p has type T is denoted as $T(a^p)$. For pointer constraints, we introduce boolean variables predicating merely on offsets. In particular, $T(*((a+f)^p)\&\mathit{0x0fff})$ represents that there is at least one physical page that has a type T instance at the page offset (the least 12 bits) of the pointer target at location $(a+f)^p$. We call such boolean variables the *offset variables* and the previous variables considering both offsets and page IDs the *location variables*.

We further introduce pointer constraints that are an implication from an offset variable to the disjunction of all the location variables with the same offset, to express the "there is at least one" correlation. The probability of the constraint is not 1.0 as it is likely there is so such a physical page if the page has been re-allocated and overwritten. Ideally, the probability is reversely proportional to the duration between the process termination and the forensic analysis. In this paper, we use a fixed value δ to represent that we believe in δ probability such a remote page is present. With pointer constraints, we are able to construct an FG that connects variables in different physical pages and perform global inference such that probabilities derived from various places can be fused together.

Example. Let's revisit the example in Section 3.2. Regular variables x_1, x_2, and x_3 now denote $utmplist(a^p)$ for a given page offset a, $P_{next}((a+384)^p)$, and $P_{prev}((a+388)^p)$, respectively. Superscript p can be considered as the id of the physical page. Offset variables y_1 and y_2 represent $utmplist(*((a+384)^p)\&\mathit{0x0fff})$ and $utmplist(*((a+388)^p)\&\mathit{0x0fff})$. Constraint C_1 (i.e., Equation (2)) is extended to the following.

$$x_1 \rightarrow x_2 \wedge x_3 \wedge y_1 \wedge y_2 \tag{14}$$

The probability of f_{C_1} remains 1.0. Assume we have three physical pages p, q, and r in DIMSUM's memory page input. Let $b = *((a+384)^p)\&\mathit{0x0fff}$ and $c = *((a+388)^p)\&\mathit{0x0fff}$, the page offsets of the pointers stored at $(a+384)^p$ and $(a+388)^p$.

Let x_4, x_5 and x_6 denote $utmplist(b^p)$, $utmplist(b^q)$, and $utmplist(b^r)$, respectively; and x_7, x_8 and x_9 denote $utmplist(c^p)$, $utmplist(c^q)$, and $utmplist(c^r)$. These variables are created when typing pages p, q and r. The pointer constraints are thus represented as follows.

$$(C_5) \quad y_1 \rightarrow x_4 \vee x_5 \vee x_6 \tag{15}$$

$$(C_6) \quad y_2 \rightarrow x_7 \vee x_8 \vee x_9 \tag{16}$$

The factor for C_5 is defined as follows.

$$f_{C_5}(y_1, x_4, x_5, x_6) = \begin{cases} \delta & \text{if } (y_1 \rightarrow x_4 \vee x_5 \vee x_6) = 1 \\ 1-\delta & \text{otherwise} \end{cases} \tag{17}$$

Recall δ reflects our overall belief of the completeness of the input memory pages. Factor f_{C_6} can be similarly defined and hence omitted. Fig. 6 presents the FG enhanced with the pointer constraint C_5. Observe that while many constraints (e.g., primitive constraints) are local to a page, the pointer constraint C_5 and the enhanced structural constraint C_1 correlate information from multiple pages. For instance, the probability of x_5 in page q can be propagated through the path $x_5 \Rightarrow f_{C_5} \Rightarrow y_1 \Rightarrow f_{C_1} \Rightarrow x_1$ to the goal variable x_1. The probability of x_1, which is the fusion of all the related probabilities, indicates if we have an instance $\texttt{utmplist}$ at the given address a.

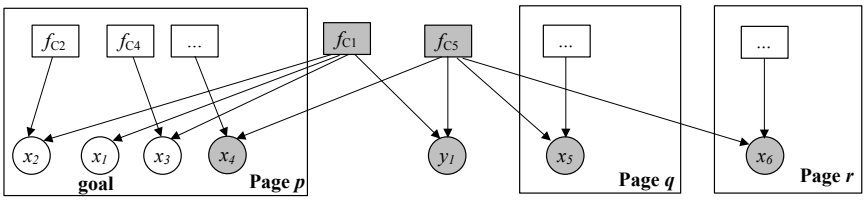


Figure 6. The factor graph enhanced with a pointer constraint. Constraints C_3 and C_6 are elided for readability. The modified part is highlighted. Constraints and variables local to a page are boxed.

4.4 Same-Page Constraints

We observe that the values of multiple pointer fields may imply that the points-to targets are within the same page. For instance, in `struct utmplist` in Fig. 2, if the higher 20 bits of the addresses stored in fields $a + 384$ and $a + 388$ are identical, we know their points-to targets must be within the same page. Hence, if we observe field $a + 384$ in page q and $a + 388$ in page r hold instances of `utmplist`, they should not be considered as support for a in p holds an instance of `utmplist`. We leverage same-page constraints to reduce false positives.

If the values of multiple pointer fields are within the same page, these pointers should not have individual pointer constraints. Instead, we introduce a joint pointer constraint that dictates the objects being pointed to by the pointers must reside in the given offsets of the same page. In our running example, the structural constraint in Equation (14) is changed to the following.

$$x_1 \rightarrow x_2 \wedge x_3 \wedge y_{1\cdot 2} \tag{18}$$

Variable $y_{1\cdot 2}$ represents a joint offset variable. It represents that there is at least one physical page that has `utmplist` instances at offsets specified by $b = *((a + 384)^p)\&0x0fff$ and $c = *((a + 388)^p)\&0x0fff$. The joint constraint is hence the following.

$$y_{1\cdot 2} \rightarrow (x_4 \wedge x_7) \vee (x_5 \wedge x_8) \vee (x_6 \wedge x_9) \tag{19}$$

For other constraints, such as the *semantic constraint* that leverages the semantics of data structures and *staged constraint* that basically performs preprocessing of input pages, their details are presented in Appendix A.

5 Evaluation

DIMSUM is implemented using Infer.NET [20], a development framework for probabilistic inference. The system constructs constraints from input memory pages, using C# and the modeling API. Solving the constraints will compute the value of each random variable, which denotes the likelihood of a memory location having a data structure instance of interest. In this section, we present the evaluation results for DIMSUM using applications on Linux and Android platforms.

5.1 Experiment Setup

Our evaluation considers dead memory pages as described in Section 1. Such dead pages come from terminated processes. The virtual memory mapping information is no longer available. In other words, DIMSUM takes a set of (dead) physical pages, and identifies target data structure instances in them.

To enable the evaluation, we have to first collect the ground truth so that we can compare it with the results reported by DIMSUM to measure false positives (FP) and false negatives (FN). We extract the ground truth in two steps: The first step is to extract data structure instances from the application process' virtual space via program instrumentation. In particular, given a data structure of interest, we instrument the program to log allocations and de-allocations of that data structure. Then, upon process termination, we visit the log file to identify the data structure instances that have been deallocated but not yet destroyed. These instances form the ground truth. The second step is to find the physical residence pages of these instances using page mapping information. The second step is needed as DIMSUM operates directly on physical pages. We implement the ground truth extraction component in QEMU [12] and an Android emulator (based on QEMU as well). Specifically, we trap the system call `sys_exit_group` to perform the extraction. Note that, on the Android platform, executables are in the form of byte code and their execution is object oriented. We have to tap into the emulator to translate object references to memory addresses.

We use DIMSUM to analyze *dead* memory pages in the system [2]. To acquire all dead pages across the system, we

[2]Live memory forensics is outside the scope of this paper. It can be achieved by techniques guided by page mapping such as KOP [6] and SigGraph [16].

Data of Interest	Benchmark Program	Size	#Input Pages	#True Inst.	Value-Invariant			SigGraph$^+$			DIMSUM		
					#R	FP%	FN%	#R	FP%	FN%	#R	FP%	FN%
Login record utmp	last 2.85	392	27266	8	48	83.3	0.0	6	0.0	25.0	8	0.0	0.0
			18186	6	46	87.0	0.0	2	0.0	66.7	6	0.0	0.0
			8898	0	40	100.0	0.0	0	0.0	0.0	1	100.0*	0.0
Browser Cookies	w3m 0.5.1	80	31303	23	93	76.3	0.0	35	34.3	0.0	22	0.0	4.3
			20848	23	93	76.3	0.0	35	34.3	0.0	22	0.0	4.3
			10423	0	70	100.0	0.0	9	100.0	0.0	9	100.0*	0.0
	chromium 8.0.552.0	44	45308	25	89	71.9	0.0	82	69.5	0.0	45	44.4	0.0
			30205	19	61	68.9	0.0	56	66.1	0.0	38	50.0	0.0
			15103	9	49	81.6	0.0	43	79.1	0.0	16	43.8	0.0
Address Book	pine 4.64	144	33186	124	1216	90.3	4.8	229	48.5	4.8	101	0.0	18.5
			22123	96	1174	92.2	2.1	174	50.1	10.4	79	0.0	17.7
			11063	63	1142	94.5	0.0	88	56.8	39.7	42	0.0	33.3
	Sylpheed 3.0.3	48	46504	309	412	25.0	0.0	412	25.0	0.0	323	5.0	0.6
			31002	204	244	16.4	0.0	244	16.4	0.0	194	0.0	4.9
			15502	92	128	28.1	0.0	128	28.1	0.0	82	0.0	10.9
Contact List	pidgin 2.4.1	60	58743	300	491	38.9	0.0	485	38.8	1.0	297	0.0	1.0
			39163	198	259	23.6	0.0	254	22.8	1.0	196	0.0	1.0
			19580	98	130	24.6	0.0	126	23.0	1.0	97	0.0	1.0

Table 3. Summary of evaluation results with data structures in user-level applications on Linux

enhance QEMU to traverse kernel data structures such as memory zones and page descriptors.

To emulate the scenario where some dead pages – especially those containing data structures of interest or their supporting data (e.g., those data structures that are pointed to by pointers in the data structure of interest) – are reused for new processes, we vary the number of dead pages provided to DIMSUM. In our experiments, we study three settings: 33%, 67%, and 100%. For example, 33% means that we randomly select 33% of the dead pages as input to DIMSUM.

Comparison with value-invariant and SigGraph We also compare DIMSUM with other techniques that can be adopted for un-mappable memory forensics. The first technique to compare with is a value invariant approach similar to the approaches in [4, 11, 25], which leverage field value patterns to identify data structure instances. The patterns we use are mainly the value patterns of pointers and those derived from domain knowledge.

The second technique to compare with is a variant of SigGraph [16]. SigGraph is a brute force memory scanning technique. It leverages the points-to relations between data structures and uses a points-to graph rooted at a data structure as its signature for scanning. *Note that the original SigGraph relies on page mappings to traverse pointers and thus cannot be applied to our "un-mappable memory" scenario.* We implement a variant of SigGraph, called SigGraph$^+$, which tries to aggressively traverse pointers even without page mappings. In particular, during scanning, SigGraph$^+$ identifies the page local offset (the lower 12 bits) of a pointer value, say x, and then tries to look for a match at offset x among all input pages. For instance, assume that (1) the graph signature of a type T is that its field f points to an object of type T_1 and (2) the page offset of the pointer value at field f is x. As along as it can find at least one page whose offset x is an instance of T_1, SigGraph$^+$ will report that an instance of T is identified.

5.2 Effectiveness on Linux Platform

In this section, we present the experimental results of applying DIMSUM to discover (1) user login records, (2) browser cookies, (3) email addresses, and (4) messenger contacts from Linux applications. A summary of these results is presented in Table 3. The specific data structures of interest, the applications, and the size of the target data structures are reported in the 1^{st}, 2^{nd}, and 3^{rd} columns, respectively. The 4^{th} column reports the total number of input pages provided to DIMSUM, and the 5^{th} column shows the total number of true instances. We compare DIMSUM with value-invariant and SigGraph$^+$. Columns "#R", "FP%" and "FN%" report the total number of instances identified by the corresponding approaches, the False Positive (FP), and False Negative (FN) rate, respectively.

From this table, we make the following observations: (1) Value-invariant has high FPs and very low FNs, (2) SigGraph$^+$ has high FPs as well, and low FNs, (3) DIMSUM has significant less FPs and low FNs. On average, the FPs for value-invariant, SigGraph$^+$, and DIMSUM are 65.5%, 38.5%, and 19.0%, respectively; the FNs are 0.4%, 8.3%, and 5.4%. Note the real FP rate of DIMSUM may be lower than the reported number because the two 100% false positive cases (those with superscripts in Table 3) can be easily pruned because the absolute value of the probability is very low (below 0.5). More details will be discussed in the case study. Precluding these two cases, DIMSUM has only 8.0% FP.

5.2.1 A Case Study

We further zoom in on one case to concretize our discussion. In the study of utility program `last`, we acquired 8 true instances and 27266 input pages, including the 2 pages that contain the 8 true instances.

The detailed results with the three different settings are presented in Fig.s 7(a), 7(b), and 7(c), respectively. Note in these figures, the X-axis represents the page offset within a physical page. For DIMSUM, the Y-axis represents the probability of a match. For value-invariant and SigGraph$^+$, since there is no probability involved, we just add "V" and "S" to the Y-axis to show their results. Also, in these figures, a ground truth is marked with $\times$.

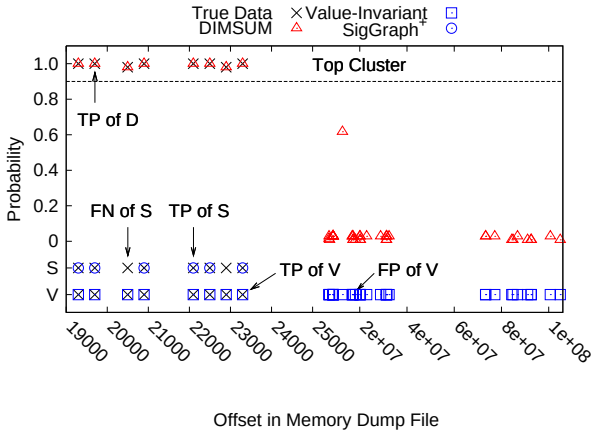
A data point marked with *both* $\times$ and the symbol of the technique means the technique identifies a true positive (TP). For example, the data point marked with *both* $\times$ and Δ as indicated in the top cluster in Fig. 7(a) is a TP for DIMSUM. A point with only a technique symbol indicates a false positive (FP). For example, the nodes in the right bottom of Fig. 7(a) are FPs for the value-invariant approach. *Note that DIMSUM only reports nodes in the top cluster*. Hence, those DIMSUM data points that are not in the top cluster are not FPs, even though they are not marked with $\times$. A point with only $\times$ indicates a FN. For DIMSUM, any single $\times$ symbols that are not in the top cluster are FNs.

When 100% dead pages are provided to DIMSUM (Fig. 7(a)), DIMSUM successfully identifies all ground truth without any FPs or FNs – in the top cluster of points whose probability is greater than 0.95. SigGraph$^+$ identifies 6 instances with 25% FN, and the value-invariant approach identifies 48 instances with 83.3% FP. Next, we randomly select 67% of the dead pages. One page containing 2 true instances is precluded as the result of the random selection. The result is shown in Fig. 7(b). DIMSUM identifies all remaining 6 true instances in the top cluster. In contrast, SigGraph$^+$ in this case identifies only 2 true instances, because for the other 4 instances, their graph signatures are not complete due to the missing pages. In contrast, *DIMSUM is able to survive as it aggregates sufficiently high confidence from the fields in the remaining 67% pages.* Finally, when 33% dead pages are analyzed, all the true instances are precluded. DIMSUM identifies one instance in its top cluster as shown in Fig. 7(c), which is an FP. But we point out that DIMSUM in the mean time determines that the instance has only a probability lower than 0.50. The user can easily discard such results.

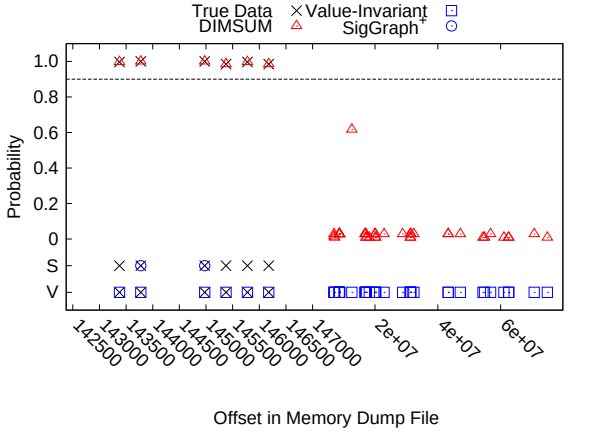
5.2.2 False Positive Comparison

In the following two subsections, we discuss the FPs and FNs of the three techniques in detail.

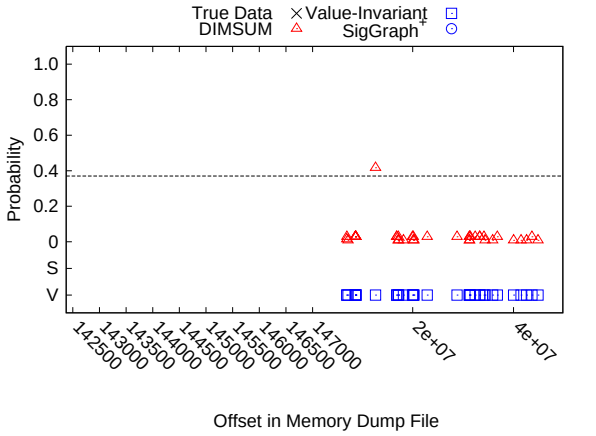
Value-invariant has high FPs because it only looks at the value patterns of the fields in the target data structure. It



(a) `last` (100% of dead pages)



(b) `last` (67% of dead pages)



(c) `last` (33% of dead pages)

Figure 7. Evaluation of DIMSUM effectiveness for discovering user login record data structure `utmp`

does not try to collect additional confidence from the child data structures (those pointed-to by the pointer fields in the target data structure). The end result is that it reports a lot of bogus data structure instances.

SigGraph$^+$ also has high FPs. Recall that, as an extension to SigGraph, SigGraph$^+$ also uses the points-to graph signature to search for instances of a data structure (Section 5.1). Given a pointer field "`T* f;`" of the data structure, it tries to confirm if `*(f)` holds an instance of `T`. However, since memory mapping is not available, `f` cannot be resolved. Instead, it aggressively looks for any instance of `T` among all pages at page offset `f&0x0fff`. The consequence is that it may find an instance which is actually not pointed to by `f`. The situation is particularly problematic when `T` is a popular type (e.g., string) so that there are instances of this type at almost any page offset. Another key reason is that SigGraph$^+$ cannot propagate probabilities across different data structures like DIMSUM to reduce FPs.

DIMSUM achieves low FP rate. As explained in Section 5.2.1, the only case (`utmp` and the 33% setting) with a 100% FP rate indeed has a very low probability, and is hence an easy-to-prune FP. The result indicates the effectiveness of DIMSUM. Probabilistic inference indeed allows global reasoning over all the connected data structures, collecting and aggregating confidence from all over the places, eventually distinguishing the true positives. The DIMSUM FPs for `chromium` (Table 3) are mainly caused by the simplicity of the cookie data structure. In other words, DIMSUM does not have a lot of sources to garner enough confidence to distinguish true positives from others. Interestingly, the 5% FPs for `Sylpheed` are mainly caused by the fact that some dead pages not from `Sylpheed` happen to have some instances that satisfy our constraints. When those dead pages are not selected (33% and 67% cases), the FPs are gone.

5.2.3 False Negative Comparison

The value-invariant technique has the lowest FNs. This is reasonable as it is the least restrictive method, admitting everything that appears to be an instance of the target data structure based on its value properties.

Both SigGraph$^+$ and DIMSUM have high FNs for the `pine` case. The main reason lies in the insufficient number of input pages, especially under the settings of 33% and 67%. In other words, the child data structures are not present in the pages provided to these techniques. Another reason for high FNs is the *cross-page* data structure instances. There are some data structure instances that span two pages. Neither technique is able to handle cross-page data structures because consecutive virtual pages do not correspond to consecutive physical pages, resulting in FNs under the 100% setting. We will leave this problem to our future work.

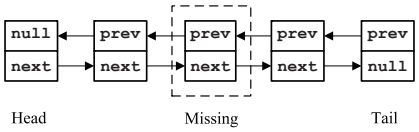


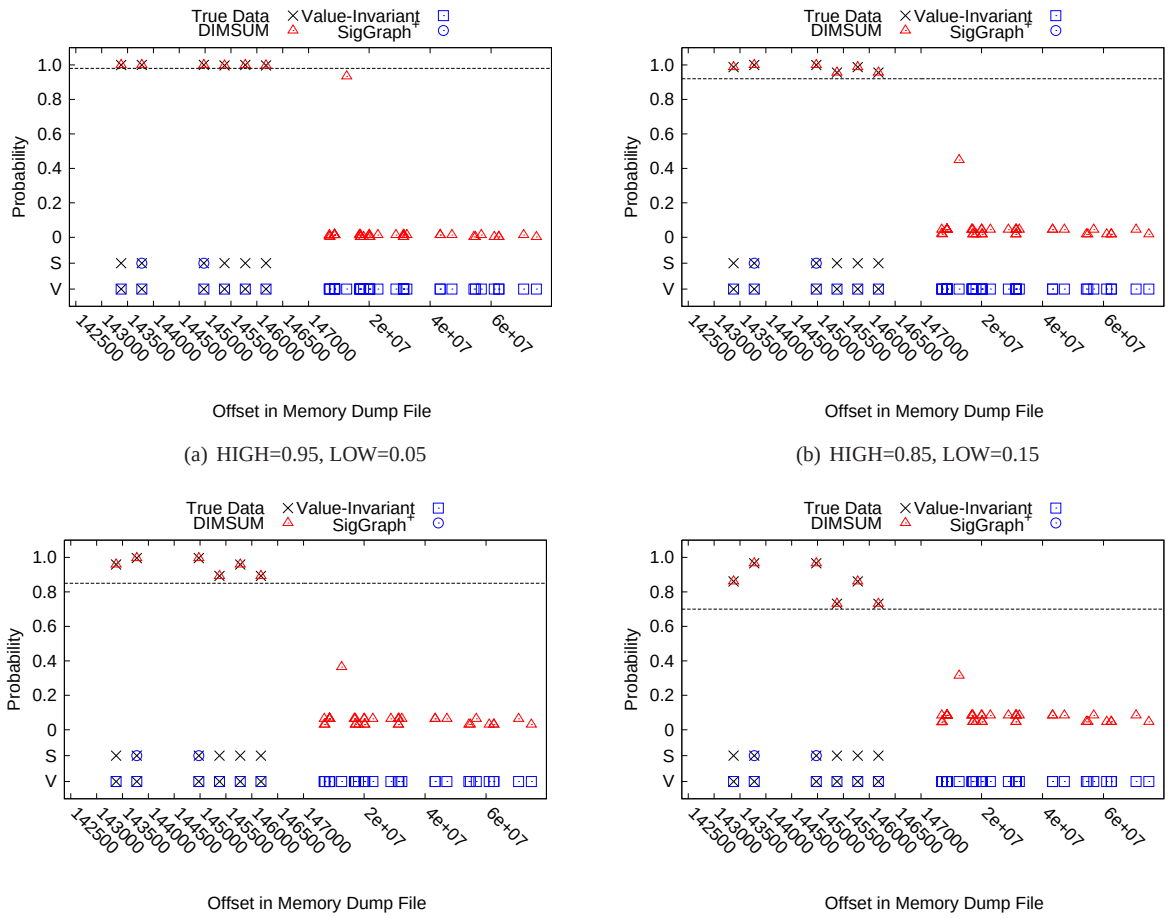
Figure 8. An abstraction of the `utmp` case. The node in the middle is missing.

In some cases, DIMSUM performs better than the less restrictive SigGraph$^+$ in terms of FNs – for example, the `utmp` structure in `last-2.85`. The main reason is that SigGraph$^+$ is based on binary reasoning, and hence a piece of memory is either an instance of interest or not. In contrast, DIMSUM does not draw binary conclusions but rather aggregates confidence gradually to form the right picture, as illustrated in Fig. 8. The whole linked list represents the `utmp` linked list (already freed). The node in the middle is missing (the page is reused). The graph signature used in SigGraph$^+$ is a node with its preceding node and succeeding node, meaning an instance of `utmp` is recognized if the `prev` and `next` pointers also point to instances of `utmp`. In this case, SigGraph$^+$ cannot recognize the head or tail due to the null pointers. And it cannot recognize the 3^{rd} node because it is missing. As a result, it cannot recognize the 2^{nd} or 4^{th} node either. In contrast, DIMSUM does not make binary judgement on individual nodes. Instead, it models them as a network of constraints. In this case, two factor graphs, one containing the 1^{st} and 2^{nd} nodes and the other the 3^{rd} and 4^{th} are formed and resolved. Aggregating probabilities in the two graphs identifies the true positives with confidence.

5.3 Sensitivity on the Threshold

So far we have a default setting for the high and low initial probabilities: HIGH=0.90 and LOW=0.10. We now study how different probability settings would affect the final results. Interestingly, we find that the results are *not* sensitive to such setting, i.e., we are able to detect roughly the same set of data structure instances under various settings. Detailed results are shown in Fig. 9, which involves the discovery of `utmp` instances under the 67% setting.

Different from Fig. 7(b), we vary the values of HIGH and LOW as follows: HIGH=0.95 and LOW=0.05, HIGH=0.85 and LOW=0.15, HIGH=0.80 and LOW=0.20, and HIGH=0.75 and LOW=0.25. The results are presented in Fig. 9(a)-Fig. 9(d), respectively. We observe that, for all four settings, the top cluster still contains the true instances. Other evaluation cases lead to similar results. As such, we



(a) HIGH=0.95, LOW=0.05

(b) HIGH=0.85, LOW=0.15

(c) HIGH=0.80, LOW=0.20

(d) HIGH=0.75, LOW=0.25

Figure 9. The Threshold Impact on the Experimental Result

conclude that the accuracy of DIMSUM is not sensitive to the probability thresholds.

5.4 Effectiveness on Android Platform

With smart phones storing a large amount of personal information, memory forensics on smart phones is increasingly important. First, mobile applications (apps) are usually long-running. Users just switch from one app to another, rarely terminating the apps. The consequence is that the trail of the user's behavior is left in the memory image instead of in permanent storage. Second, the garbage collection-based execution model decides that dead objects (and hence historic data) are not destroyed unless memory consumption exceeds capacity. In other words, if a device is not running memory intensive apps, the old data (e.g., browsing history and past GPS location) is likely to remain intact for a long period of time, favoring memory forensics. Third, users may opt to encrypt their critical information. While this makes forensics on permanent storage difficult, it has less effect on memory forensics as encrypted data have to be decrypted first in memory before usage.

DIMSUM is a particularly suitable memory forensics method for mobile devices. The reason is that on mobile platforms, dead objects are unreachable, rendering reachability-based technique such as KOP [6] inapplicable. Furthermore, garbage collector often *moves* dead pages around, making pointers invalid and disabling pointer traversing-based techniques such as SigGraph [16]. Finally, DIMSUM works directly on physical memory, which is particularly desirable when a device is locked.

Memory layout in Android Android apps are object-oriented. Programs are compiled into Dalvik byte code and run on the Dalvik virtual machine. Data structures are in the form of objects at runtime. There are some interesting memory layout features that can be leveraged by DIMSUM. According to the layout shown in Fig. 10, each object internally has a unique pointer at offset 0 pointing to

Data of Interest	Benchmark Program	Size	#Input Pages	#True Inst.	Value-Invariant			SigGraph$^+$			DIMSUM		
					#R	FP%	FN%	#R	FP%	FN%	#R	FP%	FN%
Browser Cookies	`Browser in Android-2.1`	56	32768	31	143	78.3	0.0	135	77.0	0.0	31	0.0	0.0
			21845	25	108	76.9	0.0	102	75.5	0.0	25	0.0	0.0
			10923	6	43	86.0	0.0	35	85.8	16.7	6	0.0	0.0
Phone Contacts	`Messaging in Android-2.1`	68	32768	117	283	58.7	0.0	113	0.9	4.3	117	0.0	0.0
			21845	79	182	56.6	0.0	76	0.0	3.8	79	0.0	0.0
			10923	36	101	64.4	0.0	35	2.9	5.6	36	0.0	0.0
Message Conversations	`Messaging in Android-2.1`	60	32768	102	131	22.1	0.0	100	0.0	2.0	102	0.0	0.0
			21845	60	78	23.1	0.0	59	0.0	1.7	60	0.0	0.0
			10923	40	51	21.6	0.0	39	0.0	2.5	40	0.0	0.0

Table 4. Summary of evaluation results with data structures in Android app.s

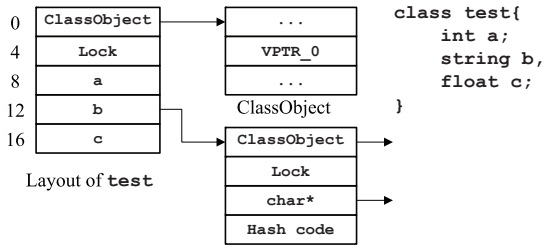


Figure 10. A Java object layout on Android.

the ClassObject (similar to the Type Information Block in other JVM implementations). A ClassObject is an array of objects that encodes the type information of a class. In other words, all objects of the same class must have a pointer field pointing to the same ClassObject. However, these pointers are *not constant* across runs as the ClassObject may be in a different location in a different run. As a result, value invariant techniques cannot identify such per-run pattern.

DIMSUM can easily model the per-run constant above. Assume the type of interest is T. Given a memory location a, x denotes $T(a)$ and the original structural constraint without modeling the ClassObject pattern is

$$x \rightarrow C.$$

C denotes a clause. The updated constraint becomes

$$x \rightarrow y \wedge C.$$

Variable y denotes $ClassObject_T(*(a))$, meaning the pointer stored at a points to the ClassObject for class T. Note that all the true positives have their ClassObject pointers pointing to the same location, suggesting their factor graphs get connected through the same random variable y. This enables their accumulative support from each other, hoisting the probability of y and eventually, distinguishing themselves from the rest.

Evaluation results We apply DIMSUM to real applications on the Android platform (version 2.1), including the Internet browser, contact manager, and message manager as they are well-known apps that handle private information.

We summarize the results in Table 4. Observe that DIMSUM does not have any FP or FN. In comparison, value-invariant has 54.2% FP rate and 0.0% FN rate on average, while SigGraph$^+$ has 26.9% FP and 4.2% FN. This is mainly attributed to the fact that DIMSUM is able to model the per-run ClassObject pattern, which identifies the corresponding objects with high accuracy.

6 Discussion

DIMSUM has several limitations. Firstly, if a program always zeros out its data right after they are used, DIMSUM cannot recover semantic information from it. This is a common limitation for many forensics techniques. In practice, however, cleaning up memory is more difficult than clearing up other types of evidence, such as screens and files. A suspect has to instrument memory management functions and intercept program exit signals. In the presence of memory swapping, he/she also has to make sure the pages that get swapped out are cleaned before exit. Moreover, if the program crashes or gets killed, cleaning up its memory may not be done in time. DIMSUM is also not effective if a target data structure is too simple, such as one with only three integer fields or just one pointer field.

Secondly, DIMSUM currently does not fully exploit value invariant properties such as "the range of an integer field of type T is $[x, y]$." Instead, it only leverages the weaker information "this is an integer field." Value invariant properties are usually acquired via profiling or domain knowledge input. We believe that DIMSUM is able to perform even better when value invariants are integrated into the constraints. In fact, DIMSUM can naturally tolerate the uncertainty of value invariant properties caused by the insufficiency of profiling runs.

Thirdly, DIMSUM currently requires users to manually write down data structure specifications following our grammar, in order to generate constraints and then factor graphs. Part of our future work is to make this process more automated.

Finally, DIMSUM currently does not handle data structure instances *across* pages. Possible solutions require

extending the inference model. Also, DIMSUM does not work for encrypted memory pages (e.g., encrypted swap page files by some OSes).

7 Related Work

Memory forensics Our work centers around memory forensics, a process of analyzing a memory image to infer earlier states of a computer system. Such process often involves discovering some data structure instances of interest in the image. The state of the art techniques fall into two main categories: memory traversal through pointers (e.g., [1, 6, 7, 17, 22]) and signature based scanning (e.g., [4, 11, 16, 24, 25]).

Memory traversal approaches attempt to build a roadmap of all data structures, starting from global objects and traversing along points-to edges. Such an approach has to resolve generic pointers such as `void*` and cannot traverse further if a pointer is corrupted. SigGraph [16] complements those approaches by deriving a context-free pointer-based signatures for data structures. These techniques mostly work for "live" objects because "dead" objects cannot be reached by memory traversal due to missing page tables and unresolvable pointers. In contrast, DIMSUM supports dead object recovery and does not require memory traversal from root variables. In other words, DIMSUM can be used to scan arbitrary memory pages.

Signature-based scanning directly searches memory using signatures. A classic approach is to search specific strings in memory. Other notable techniques include PTfinder [25] for linear scanning of Windows memory to discover process and thread structures, as well as Volatility [24] and Memparser [4] capable of searching other types of objects. These techniques rely on the presence of value invariant signatures. The quality of such signatures heavily depends on profiling runs of the subject program. If the number of profiling runs is small, the signatures tend to over-approximate the real value invariants, leading to false positives. If the number is large, there may not be any valid invariants. In contrast, DIMSUM does not rely on value invariants. Moreover, it can be combined with value invariant-based techniques to mitigate the uncertainty of value invariant profiling.

More recently, Walls et al. presented DEC0DE [29], a system also leveraging probabilistic inference for recovering information such as call logs and address books from smartphones across a variety of models. DEC0DE uses a set of probabilistic finite state machines to encode the target data structures, more specifically call logs and address books. Such models can tolerate unknown data formats. DIMSUM aims at recovering generic program data structures specified by forensics investigators.

Cryptographic key discovery ColdBoot [13] is a technique that discovers cryptographic keys from a power-off computer. ColdBoot and DIMSUM differ in their goals and enabling techniques: (1) ColdBoot focuses on uncovering keys, whereas DIMSUM uncovers general data structure instances. (2) ColdBoot exploits specific characteristics of encryption/decryption algorithms and data structures to recover keys. For instance, for RSA key discovery, ColdBoot uses field value-invariants, such as a field starting with `0x30` and the DER encoding of the next field (`02 01 00 02`), to scan the memory. In comparison, DIMSUM exploits constraints of global points-to relations between data structures and of primitive types of data structure fields. Similar to ColdBoot, another technique [19] also exploits cryptographic data structure-specific information to recover keys.

Other works using probabilistic inference Probabilistic inference has a wide range of applications, such as face recognition (e.g., [21]), specification extraction (e.g., [14, 18, 3]), and software debugging (e.g., [10]). While DIMSUM relies on the same inference engine, it faces a different set of challenges due to the different application domain. Moreover, the entailed modeling techniques are also different.

Laika [9], a data structure definition derivation system, leverages statistical analysis as well. Laika aims to derive the data structure *definitions* in a binary program. It starts with zero knowledge about the data structure definitions, and uses data instances to eventually cluster and derive data structure definitions. DIMSUM solves a completely different problem. It starts with data structure definitions and tries to find their *instances in memory*. The modeling techniques are completely different. Furthermore, Laika relies on memory mapping when traversing memory, whereas DIMSUM does not.

8 Conclusion

We have demonstrated that it is possible to discover data structure instances of forensic interest from a set of memory pages without memory mapping information. Such a capability is realized by DIMSUM, a system that employs probabilistic inference to extract the data structure instances with confidence. DIMSUM takes data structure specification and memory page content as input, builds factor graphs based on boolean constraints about the data structures and memory content, and produces data structure recognition results with probabilities. Our experiments with Linux and Android-based applications show that DIMSUM achieves higher accuracy in partial memory image analysis compared with non-probabilistic approaches.

Acknowledgement

We would like to thank Aditya Nori and the anonymous reviewers for their insightful comments and suggestions. This research was supported, in part, by the US National Science Foundation (NSF) under grant 1049303. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Mission critical linux. In Memory Core Dump, http://oss.missioncriticallinux.com/projects/mcore/.

[2] Single precision floating-point format. http://en.wikipedia.org/wiki/Single_precision_floating-point_format.

[3] N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the 2011 ACM SIGPLAN conference on Programming language design and implementation (PLDI'11)*, 2011.

[4] C. Betz. Memparser. http://sourceforge.net/projects/memparser.

[5] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[6] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 555–565, Chicago, IL, USA, 2009.

[7] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev. Face: Automated digital evidence discovery and correlation. *Digital Investigation*, 5(Supplement 1):S65 – S75, 2008. The Proceedings of the Eighth Annual DFRWS Conference.

[8] J. Chow, B. Pfaff, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole-system simulation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[9] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI'08)*, pages 231–244, San Diego, CA, December, 2008.

[10] L. Dietz, V. Dallmeier, A. Zeller, and T. Scheffer. Localizing bugs in program executions with graphical models. In *Proceedings of the 2009 Advances in Neural Information Processing Systems*, December 2009.

[11] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)*, pages 566–577, Chicago, Illinois, USA, 2009. ACM.

[12] B. Fabrice. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.

[13] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, August 2008.

[14] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI'06)*, pages 161–176, Seattle, Washington, 2006. USENIX Association.

[15] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, February 2011.

[16] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, February 2011.

[17] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, February 2010.

[18] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, pages 75–86, 2009.

[19] C. Maartmann-Moe, S. E. Thorkildsen, and A. rnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digital Investigation*, 6(Supplement 1):S132 – S140, 2009. The Proceedings of the Ninth Annual DFRWS Conference.

[20] T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, 2009. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[21] B. Moghaddam, T. Jebara, and A. Pentland. Bayesian face recognition. *Pattern Recognition*, 33(11):1771–1782, November 2000.

[22] P. Movall, W. Nelson, and S. Wetzstein. Linux physical memory analysis. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*, pages 23–32, Anaheim, CA, 2005. USENIX Association.

[23] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference (2nd ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[24] N. L. Petroni, Jr., A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197 – 210, 2006.

[25] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation*, 3(Supplement-1):10–16, 2006.

[26] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed*

System Security Symposium (NDSS'11), San Diego, CA, February 2011.

[27] J. Solomon, E. Huebner, D. Bem, and M. Szezynska. User data persistence in physical memory. *Digital Investigation*, 4(2):68 – 72, 2007.

[28] R. M. Stevens and E. Casey. Extracting windows command line details from physical memory. *Digital Investigation*, 7(Supplement 1):S57 – S63, 2010. The Proceedings of the Tenth Annual DFRWS Conference.

[29] R. J. Walls, E. Learned-Miller, and B. N. Levine. Forensic Triage for Mobile Phones with DEC0DE. In *Proceedings of 2011 USENIX Security Symposium*, August 2011.

[30] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. pages 239–269, 2003.

Appendix A: Other Constraints

A.1 Semantic Constraints

Besides the aforementioned constraints in Section 4, there could exist semantic constraints imposed by the data structure definitions. For example, a field `pid` tends to have a value ranging from 0 to 40000; an unused fields tends to have zero values. Meanwhile, it is also possible that a particular data structure field has a value invariant. As such, semantic constraints can be used to prune unmatched fields.

A.2 Staged Constraints

Discussion in Section 4 implies that we need to create many boolean variables for each memory location. In particular, for each offset in every page, we introduce variables to predicate on its various primitive types and types of interest. Constraints are introduced among these variables, describing any possible dependencies. The order of introducing the constraints is *irrelevant*. The entailed FG is often very large and takes a lot of time to resolve. We develop a simple preprocessing phase to reduce the number of variables and constraints. In particular, we first scan each input page and construct primitive constraints, describing if each offset is an integer, a char, a pointer, etc. In the second step, we construct structural and other constraints. We avoid introducing a variable predicating on if a base address a is of type T if any of the corresponding field primitive constraints has a LOW probability. We leverage the observation that such inference is simple and does not need FG to proceed.