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

***Abstract*—Discovering the semantic data of interest in memory without mapping information is a desired goal in memory forensics. Such un-mappable memory could be the entire free pages of the system, the memory swap file, or a corrupted memory dump. However, existing road-map based memory forensics techniques cannot work in these cases as the pointers in the un-mappable memory cannot be resolved and navigated. To address this problem, this paper presents a probabilistic inference based approach DIMSUM to enable the discovery of semantic data of interest from un-mappable memory. Given a set of memory pages, and the specifications of the target data structure, DIMSUM is able to identify instances of the data structures in these pages with confidence. Specifically, it automatically builds graphical models based on the constraints dictated in data structure specifications and the observed values, and then translates the constraints into factor graphs, on which probabilistic inference is performed to finally generate the result with probability rankings. Experiment shows DIMSUM achieves significant better result than existing approaches in un-mappable memory analysis.**

I. Introduction

A common task in computer forensics is to uncover semantic data of interest, such as user names, passwords, contact lists, chat content, IP addresses, cookies, and web browsing history from raw memory. A number of recent efforts have demonstrated the capability of uncovering instances of data structures defined in an application or the operating system kernel. However, the existing solutions critically depend on *memory mapping information*. For example, KOP [5], REWARDS [15] and SigGraph [14] all require that pointers between data structures 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 variables or variables on stack frames.

Unfortunately, such memory mapping information is not always available. Yet it is desirable that a computer forensics investigator be empowered with 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 II). While these "dead" memory pages may contain data of forensic interest, existing memory mapping-based forensics techniques (e.g., [5], [14], [15]) 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.

In addition to the above scenario of "dead pages left by a terminated process", there are other common computer forensics scenarios that require analyzing partial memory image in the absence of memory mapping information. For example, after a sudden power-off of a computer, a subset of the memory pages belonging to a previously running process may exist in the disk due to page swapping. But the memory mapping information maintained by the OS kernel for that process is lost after the power-off. As another possibility, due to the fact that most existing memory forensics techniques depend on memory mapping information and on the completeness of a process' memory image, countermeasures may be taken by adversaries to inflict digital or even physical damages to the memory image of a computer. For example, it has been shown that advanced kernel-level attacks can be launched to disable the recovery of critical kernel objects from a memory image [14]. And some of those kernel objects contain memory mapping information for application processes.

Motivated by the forensics needs discussed above, we develop a new system, called DIMSUM[1], which is capable of uncovering semantic data instances of forensics interest from a set of memory pages without memory mapping information. In particular, DIMSUM remains effective even with an incomplete subset of memory pages of an application process. As such, DIMSUM differs from, and complements most existing approaches to memory forensics (e.g., [1], [3], [5], [6], [9], [14], [15], [19], [21], [22], [23]), where their primary focus is on extracting semantic information from *live* memory (either on-line or off-line). Many of these efforts (e.g. [1], [5], [14], [15], [19], [21], [22], [23]) rely on certain memory mapping information – such as the system symbol table – to search for variables and data structure instances in the memory that can be reached directly or indirectly (e.g., by following the pointers between variables such as in KOP [5] and SigGraph [14]). We also note that some of the existing approaches (e.g., [3], [9], [21], [23]) also leverages value-invariant signatures of data structures (e.g., "data structure field x having a special value

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

or value range"). These techniques are effective if unique signatures can be generated for the subject data structures. Unfortunately, such a signature may not always exist for a data structure. The semantics of the data structure may dictate that a field has highly diverse, unconstrained values.

DIMSUM is based on *probabilistic inference*, which is widely used in computer vision, expert system, error code correction, and software debugging (e.g., [12], [16], [17], [20], [26]). 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, our system automatically builds graphical models from the data structure specification and the observed values in the input pages, and translates it into *factor graphs* [26], on which probabilistic inference will be carried out to extract target data structure instances with confidence, quantified with probabilities.

The salient features of DIMSUM are the following: (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 hostile 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 and demonstrated the effectiveness of DIMSUM.

II. Background and System Overview

A. Observation

DIMSUM was first motivated by the "dead memory pages left by terminated process" scenario as described in Section I. More specifically, we notice that, when a process is terminated, neither Windows nor Linux operating system clears the content of its memory pages. We believe one of the reasons is to avoid memory cleansing overhead. Moreover, Chow et al [7] 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 happen before the shredding operation, leaving some sensitive data in the dead memory pages.

Second, we also observe that dead memory pages may remain intact for a non-trivial duration, which we call their *death-span*. In an experiment, we observe that the death-span of the 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 as in our study, the death-span of dead pages may be even longer. Similar study on the age of freed user process data in Microsoft Windows XP (SP2), by Solomon et al [24], 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 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) very quickly. The much shorter dead-span of kernel objects (typically in 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 [4], which uses LIFO as the memory recycling policy; whereas memory pages of processes are managed by the buddy system [4] which groups memory frames into lists of blocks having 2^k contiguous frames, and hence page frames tends to have longer dead-span.

We point out that the above scenario is *not the only one* that involves memory pages without memory mapping information. Two other examples are described in Section I (i.e., "swapped-out pages after power-off" and "memory image after malicious kernel object manipulation"). Another interesting scenario is demonstrated in Coldboot [11]: After a machine with modern DRAM is powered off, the content of the DRAM will disappear *gradually* instead of immediately, making it possible to obtain partial memory image with no or partial memory mapping information.

B. Challenges

Our observations above motivate the development of DIMSUM. Compared with existing approaches, DIMSUM raises 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 page belongs to which process, let alone the association of a physical page with the virtual address space of a particular process. Even if we can identify some pointers in the pages, we still cannot follow those pointers without the address mapping information. Moreover, it is possible that a data structure instance straddles two virtually contiguous but physically non-consecutive pages, making it difficult to identify that instance without page 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. 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 symbolic information is available, this memory location can be typed through its reference path (as in [5]). To DIMSUM, however, such information is not provided.

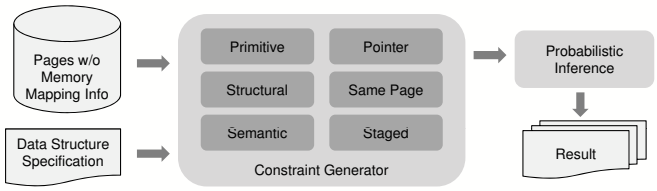


Figure 1. Overview of DIMSUM.

C. DIMSUM Overview

To address the above challenges, we take a *probabilistic inference* and *constraint solving* approach in the design of DIMSUM. 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 such as field offset and type information, which can be obtained from either application documentation, debugging information, or binary reverse engineering [15], [13].

One of the key components 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 (to be detailed in Section IV).

Next, the *probabilistic inference* component automatically transforms all the constraints into a factor graph [26], 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.

III. DIMSUM Design

The essence of our technique behind DIMSUM is to formulate the data structure recognition problem as a probabilistic constraint solving problem. In this section, we first use a working example to demonstrate the basic idea, which relies on solving boolean constraints. Then we discuss the practical issues in Section III-A, and present how we model our problem and how the underline probabilistic inference technique works in Section III-B.

Ideally, our technique takes (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 tries 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 I and assuming a 32 bit machine, the generated constraint for the `utmplist` structure would be:

```
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.

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

Note that the subscripts are used to denote field names. Intuitively, the above formula means that if the location starting at a denotes an instance of `utmplist`, the location at a contains an integer, the location at $a+4$ contains an integer as well, $a+8$ contains a char 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 their domain knowledge.

Besides the above constraints, ideally, 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, 12, 16 have been determined to contain integer, integer, non-negative integer, char array with size 16, the primitive constraints $I(0)$, $I(4)$, $I(8)$, $C(12)[16]$ (see Table I for definitions) are generated. The constraints for other primitive types are similarly 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 $C(a+8)[32]$ is not satisfied. In contrast, $a=4$ may be a satisfying

Predicate	Definitions
$\tau(x)$	The region starting at x is an instance of a user-defined type τ
$I(x)$	The location at x is an integer.
$F(x)$	The location at x is a floating point value.
$D(x)$	The location at x is a double floating point value.
$S(x)$	The location at x is a string.
$C(x)$	The location at x is a char.
$P(x)$	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 I
PREDICATE DEFINITIONS USED THROUGHOUT THE PAPER.

valuation.

A. Practical Problems

However, the basic design faces a number of practical problems in the context of DIMSUM. In particular:

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

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

Incompleteness: Due to incompleteness (Section II), we may see only part of a data structure. Our system should be able to resolve constraints even for such a partial structure.

B. Probabilistic Inference

To address the above issues, we formulate the problem as a probabilistic inference problem and use a technique called *belief propagation* (BP) [20], [26]. Beliefs are associated with individual constraints, representing the user's view of uncertainty. They are efficiently propagated and aggregated over a graphical representation called *factor graph* (FG) [26]. The probabilities of boolean variables can hence be queried from the FG. Next we use an example to explain the concept of FG and BP.

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 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 the given 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).

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

In the context of 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 $a + 384$, often time, we can only say that it is likely a pointer. Moreover, if the structural pattern of $T_{utmplist}$ is not unique, we can similarly assign a probability to constraint (3) according to structural isomorphism [14].

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. Consider 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 the following probabilistic functions.

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

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

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

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

With these probabilistic constraints, the joint probability function is defined as follows [20], [26].

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

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

In particular, Z is the normalization factor [20], [26].

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 II
BOOLEAN CONSTRAINTS WITH PROBABILITIES.

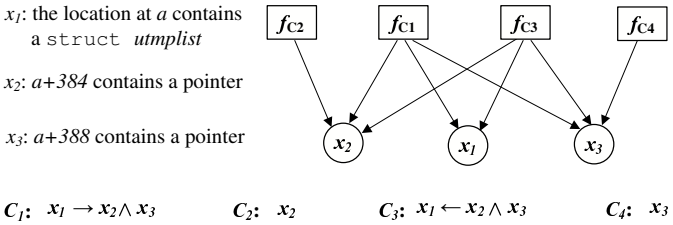


Figure 3. Factor Graph Example.

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.

Consider the previous example. Table II 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 constrained probability 0.8 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 value in f_{C_2} is what we believe and it reflects only a local view of the constraint, whereas the computed value represents a global view with beliefs over the entire system being considered. Hence, the technique is also called *belief propagation* [20], [26].

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 entitled by Equation (11) could be very expensive because it has to enumerate the combinations of variable valuations. *Factor graph* [20], [26], [16] 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 for the previous example. The *sum-product* algorithm [20], [26] can leverage factor graphs to compute marginal probabilities in a highly efficient way. The algorithm is iterative. In particular, beliefs are propagated between adjacent nodes through message passing. The beliefs of a node are updated by integrating the messages it receives. The algorithm terminates when the beliefs become stable. At a high level, one can consider beliefs as energy applied to a mesh such that the mesh transforms to strike a balance and minimize free energy. Belief propagation (BP) has a wide range of successful applications in artificial intelligence, information theory and debugging [16], [12]. In this paper, we focus on using BP for DIMSUM and we leverage an existing BP package called *Infer.NET* [17].

In order to conduct probabilistic reasoning using 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 variables needed by reducing m. Then constraints are introduced. Constraints are essentially boolean formula on the boolean variables. Beliefs are assigned to these constraints to express uncertainty. Constraints and beliefs are programmed as scripts for *Infer.NET*. The *Infer.NET* engine translates the constraints into FGs and conducts the inference. After the inference,

we can identify data structure instances by querying the probabilities of the corresponding boolean variables. We often report those within the highest probability cluster to the user of DIMSUM.

IV. Generating Constraints

This section presents the suite of constraints used in DIMSUM. They fall into the following categories: *primitive constraints* that associate beliefs 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 beliefs. They are conjoined and solved by the inference engine.

A. Primitive Constraints

Primitive constraints allow us to assign beliefs to boolean variables. Sample primitive constraints are f_{C_2} and f_{C_4} in Equation (6) and (8) in Section III. 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`. In other words, for each memory location, we introduce a boolean variable and the corresponding primitive constraint for each of these types if a data structure of interest involves these types.

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

String: To decide the belief of a string (a `char` array), we inspect the bytes starting with the given location. Firstly, a string is ended with a NULL byte. Secondly, a string often contains the printable ASCII ($[32, 126]$) or some special characters such as carriage return (CR), new-line (LF), and tab (Tab). If the two conditions are satisfied, the string constraint is set to a HIGH belief, and other primitive constraints are set to LOW. It is possible that the bytes starting at x look both like a string and like an integer. A unique advantage of BP (belief propagation) is that we can assign HIGH beliefs to multiple primitive constraints on x. Intuitively, it means we believe it could be multiple types. Note that *a belief is not a probability such that the beliefs of all the primitive constraints of a location x do not need to sum up to 1*. Assigning multiple HIGH beliefs allows x to be eligible in playing different roles during inference and we do not need to make the decision upfront on if x is a string or an integer. The inference process will eventually decide if it is a string or an integer, by considering beliefs from other parts of the FG.

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.

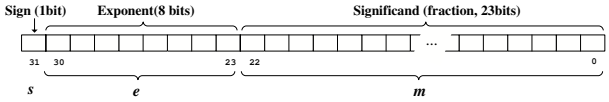


Figure 4. Float Point Representation.

Now if we examine the value of a float 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 `float` 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 float 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 belief of $F(x)$ to LOW, otherwise HIGH. Double type is handled similarly. The details are elided.

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 belief to HIGH for all these primitive constraints. Fortunately, a `char` field is mostly padded. Hence, we can limit our test to offsets aligned with the word boundary. More particularly, we only assign a HIGH belief to locations whose four bytes values fall into $\{0, 255\}$.

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

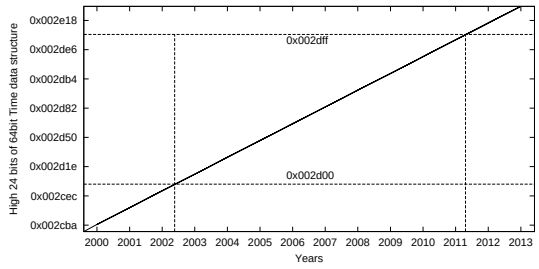


Figure 5. Common High Bits in a Time Data Structure.

property that high bits are less frequently updated than lower bits. It allows us to create constraints to infer time data structures by using common bit fields for all time values during a given period.

For example, Fig. 5 shows the values of 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, high 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.

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 beliefs. For instance, it is often possible to find out from the data structure specification that an integer timeout field must have the value within 0-2^{10}. We could use such semantic information to assign LOW belief to values outside the range.

Lastly, zeros present an interesting case for us because it could have multiple meanings: an integer with the value 0; an empty string; a null pointer; and so on. We assign HIGH beliefs to all these types except for cases in which the fields in vicinity are also having zero values. 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 belief is set to LOW. In general, the belief is reversely proportional to the length of consecutive zeros.

B. Structural Constraints

DIMSUM takes the specification of a set of data structures as input. The specification includes the field offsets and field types of the data structures that are of interest. For instance, if we are interested in instances of data structure T, and 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 into the factor graph automatically.

A structural constraint is intended to denote the dependence that as long as a given location x is an instance of T, then the x's offsets must be of the fields types described by T's specification. An example of such constraint was introduced in Equation (1) in the beginning of Section III. 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 primitive field types. These variables were introduced in the previous step when primitive constraints were generated. A sample factor graph after such process is the subgraphs rooted at f_{C_1} in Fig. 3. Since the constraint is always certain, meaning as long as x is of T type, its offsets must follow the structure dictated by T's definition. The belief of structural constraints are always 1.0 (see Equation (5) in Section III).

C. 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 T type. In particular, we will add boolean variables $T(*(a+f))$ to the structural constraint. Note that T could belong to primitive types, user defined types, or function pointers. 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 typed 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.

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. The detailed solution is presented in the following.

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 variable predicates merely on offsets. In particular, $T(*((a+f)^p)\&0x0fff)$ 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 belief of the constraint is not 1.0 as it is likely there is not such a physical page if the page has been re-allocated and overwritten. Ideally, the belief is reversely proportional to the duration between the process termination and the 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 beliefs derived from various places can be fused together.

Example. Let's revisit the example in Section III-B. Regular variables x_1, x_2, and x_3 now denote $utmplist(a^p)$ for a given address 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 corresponding factor f_{C_1} remains the same. That is, the belief is 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 belief regarding 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 beliefs, indicates if we have an instance `utmplist` at the given address a.

D. 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 $b = *((a+384)^p)\&\mathit{0x0fff}$ and $c = *((a+388)^p)\&\mathit{0x0fff}$. The joint constraint is hence the following.

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

E. Semantic Constraints

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

F. Staged Constraints

The previous discussion 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 constraints. We avoid introducing a variable predicating on if a base address a is of type T if any

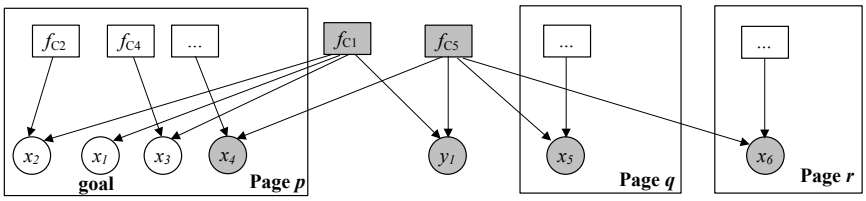


Figure 6. The factor graph enhanced with a pointer constraint. Constraints C_3 and C_6 are elided for readability. The modified part is highlighted. Constraints and variables local to a page are boxed.

```
1  using System;
2  using MicrosoftResearch.Infer.Models;
3  using MicrosoftResearch.Infer.Distributions;
4  using MicrosoftResearch.Infer;
5  public class Simple_DIMSUM_Example
6  {
7      static void Main()
8      {
9          //1. Declare Boolean Variables
10         Variable<bool> x1 = Variable.Bernoulli(0.5).Named("x1");
11         Variable<bool> x2 = Variable.Bernoulli(0.5).Named("x2");
12         Variable<bool> x3 = Variable.Bernoulli(0.5).Named("x3");
13
14         //2. Define Probabilitics Model
15         Variable.ConstrainEqualRandom<bool, Bernoulli>
16                 (!x1 | (x2 & x3), new Bernoulli(1));
17         Variable.ConstrainEqualRandom<bool, Bernoulli>
18                 (x2, new Bernoulli(0.9));
19         Variable.ConstrainEqualRandom<bool, Bernoulli>
20                 (!(x2 & x3) | x1, new Bernoulli(0.8));
21         Variable.ConstrainEqualRandom<bool, Bernoulli>
22                 (x3, new Bernoulli(0.9));
23
24         //3. Create an Inference Engine
25         InferenceEngine ie = new InferenceEngine();
26         ie.ShowFactorGraph = true;
27
28         //4. Compute Marginal Probabilities
29         Console.WriteLine("x1: " + ie.Infer(x1));
30     }
31 }
```

Figure 7. Sample code on using Infer.NET to model C_1 - C_4 in our working example and compute $p(x_1)$.

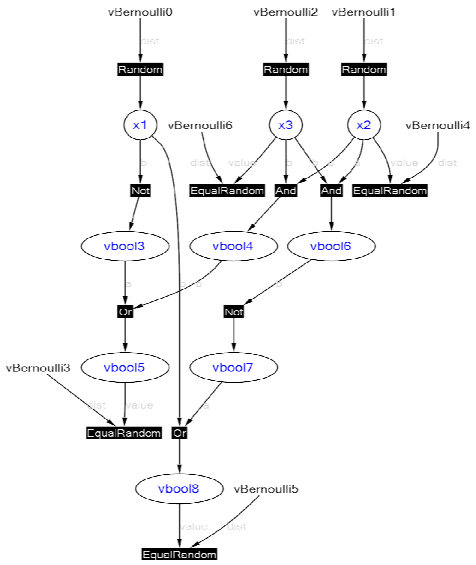


Figure 8. Factor graph of the example code from Infer.NET. Note each variable is denoted as a circle and each factor or constraint as a square. If a variable participates in the factor or the constraint, then an edge is shown between the corresponding circle and square [17].

of the corresponding field primitive constraints has a LOW belief. We leverage the observation that such inference is simple and does not need FG to proceed.

To identify data structures crossing page boundaries, we construct partial structural constraints and use them to search the heads and tails of pages. For instance, assume an `utmplist` instance is evenly split into two parts by a page boundary. To identify the first half, we will create a structural constraint modeling the first half of the data structure and try to resolve it at the end of a page.

V. DIMSUM Implementation

In this section, we present details of DIMSUM's implementation, in particular about how we generate constraints and conduct the probabilistic inference.

The key part of DIMSUM is the probabilistic inference component. We use Infer.NET, a framework for belief propagation with factor graphs as its internal model [17]. To use such framework, we do the following: (1) declare the boolean variables associated with each candidate memory cell, (2) define the corresponding constraints using their modeling API in C# code, (3) create an inference engine, and (4) execute the inference query over the boolean variables of interest.

We use our working example (Equation [2-8]) in Section III-B) to demonstrate the process. As illustrated in Fig. 7, we declare the three boolean variables $x1$, $x2$, and $x3$, initially with bernoulli value 0.5 (meaning they have 50% possibility of being the instances as we have no observations yet). Lines 15-16 model Equation (5). Similarly, Equation (6), Equation (7) and Equation (8) are modeled from line 17 to line 22. After that, we create an inference engine (line 25), and visualize the factor graph (line 26) as shown in Fig. 8. Finally, we compute the marginal probability of $x1$ (line 29), and eventually we get $p(x1) = 0.71$, which corresponds to the probability of the given address being an instance of the type of interest. Also, in this factor graph, there are a total of 9 boolean variables (represented as circle), and 13 factors or constraints (represented as square). Some variables and factors are generated internally.

In DIMSUM, we implement a compiler as the *constraint*

generator that can automatically generate variable declarations and translate constraints to C# code. The compiler is mainly pattern driven. The details are elided.

The implementation of our inference component is mainly in C#. When using our system, users need to provide the subject data structure specification and memory pages. DIMSUM then processes the pages, generates constraints, and compiles the constraints to C# code, which will further get compiled and linked with other supporting libraries from Infer.NET. Running the compiled binary delivers the final data structure instance(s) uncovered.

VI. Evaluation

This section present evaluation results on DIMSUM. We first describe our experimental setup in Section VI-A. Then we present a number of case studies of using DIMSUM to evaluate its effectiveness in Section VI-B. Finally, we report DIMSUM's performance overhead in Section VI-C.

A. Experiment Setup

We evaluate DIMSUM in the challenging forensics scenario of "dead pages left by terminated processes" (described in Section I). The dead pages come from *multiple* terminated processes without the corresponding memory mapping information.

To evaluate DIMSUM, we need to first collect the ground truth so that we can measure the false positives (FP) and false negatives (FN) of data structure instance recognition. We extract our ground truth data in two steps: The first step is to extract data structure instances from the application process' virtual memory space via program instrumentation. The second step is to locate them in the physical pages – using page mapping information. The second step is needed as DIMSUM operates on physical pages. In addition, since some of the dead pages can be re-allocated to new processes, we need to exhibit such behavior to demonstrate the robustness of DIMSUM.

Target instance collection Our data structures of interest are usually program specific, and the true (target) data structure instances are dynamically generated during program execution. Thus, we instrument the program source code, in particular the corresponding object allocation and deallocation sites, to collect the ground truth data at the process level. Then, when the process exits, we visit the log file to identify them: the truth data are those that have been deallocated but not yet overwritten. We also need to identify the corresponding physical pages where the target data reside. We call these pages "target" pages. We have to identify the target pages right before the process terminates because its page mapping will be destroyed after termination. We implement this component in the QEMU [10] system emulator. Specifically, we trap the system call `sys_exit_group` to traverse kernel data structures and acquire all the pages belonging to the dying process.

Input page collection After the termination of the process, its pages become dead pages. To extract all other dead pages (from other processes) in the system, our enhanced QEMU traverses kernel data structures such as memory zones and page descriptors. All dead pages will be the input to DIMSUM.

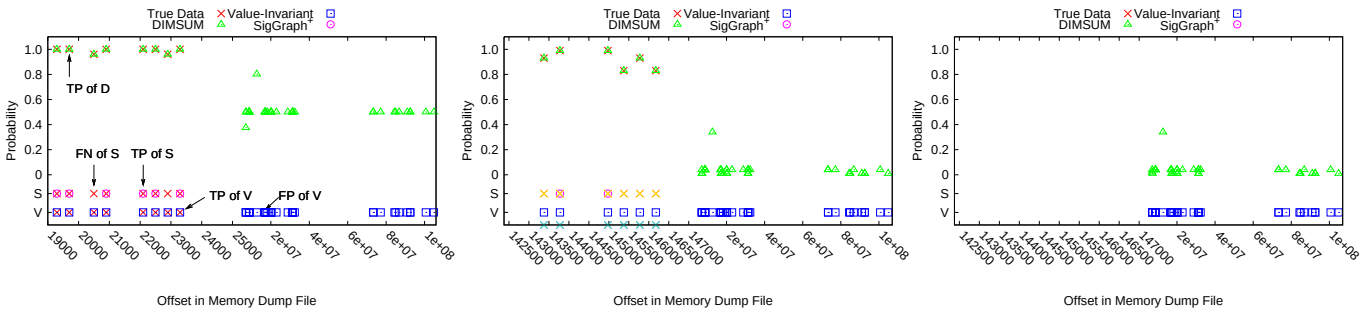
To simulate the scenario where some dead pages – especially the target pages – are reused for new processes, we vary the percentage of target pages in the input to DIMSUM. In our experiments, the percentages are 33%, 67%, and 100%. For example, 33% means that we randomly select 33% of the target pages in the input to DIMSUM.

B. Effectiveness

In this section, we present a number of case studies, to demonstrate how DIMSUM discovers (1) user login records, (2) browser cookies, (3) email addresses, and (4) messenger contacts. The specific data structures of interest and the benchmark programs are shown in the 1^{st} and 2^{nd} column of Table III.

To compare DIMSUM with other techniques, we implement a value-invariant approach, and a variant of the SigGraph [14] approach. The value-invariant approach simulates the approaches in [3], [9], [23], which leverages field value patterns to identify data structure instances. Since DIMSUM does not demand any profiling, for fair comparison, we only use the pointer value pattern of a data structure to simulate the value-invariant approach. Note that such patterns can be directly derived from the data structure specification, without profiling. SigGraph is a memory scanning technique [14]. It leverages the points-to relations of multiple data structures. It looks for the points-to graph pattern of a data structure to identify its instances. However, SigGraph is not directly applicable in our context, as traversing along a points-to edge requires knowing the target page of a virtual address, but we do not assume any page mappings. To achieve comparison, we implement a variant of SigGraph, called SigGraph$^+$. Given a virtual address, whose page offset is x, the new graph pattern search algorithm tries to traverse the offset x of all pages to look for a pattern match. For instance, assume the graph pattern of a type T is that its field f points to a type T_1. Assume the page offset (the lower 12 bits) of the pointer value at the f field is x. As along as it can find at least one page whose offset x is an instance of T_1, SigGraph$^+$ concludes that it identifies an instance of T.

1) User Login Record Discovery: The first case study is for our working example, about uncovering user login records (the `utmp` data structure shown in Fig. 2) in the entire free pages from a Linux utility program `last`. In particular, when we run `last` in our test environment, we collect 8 true instances which reside in two pages. We take



(a) `last` (100%) (b) `last` (67%) (c) `last` (33%)

Figure 9. Effectiveness Evaluation of DIMSUM for Discovering User Login Record Data Structure `utmp`.

a snapshot of the system when `last` dies, and get 27266 free pages in total including the 2 pages containing the true instances.

The experiment result is presented in Fig. 9(a), Fig. 9(b), and Fig. 9(c). Note in all of the effectiveness evaluation figures (from Fig. 9(a) to Fig. 12(c)), X-axis represents the offset in the page files. For DIMSUM, Y-axis represents the probability, and for Value-invariant and SigGraph$^+$, as there is no probability associated, we just use a V and S to stands for them in Y-axis. Also, as illusrated in Fig. 9(a) true data point is marked with $\times$, if a data point marked with both $\times$ and the corresponding point, it means a true positive (TP) data, otherwise a false positive (FP) point; if there is only a $\times$ mark, it means a false negative (FN) point for the corresponding technique. Meanwhile, there could be very crowed data points, a zoom-in may need to see the result.

When all pages (100%) are availabe to DIMSUM, shown in Fig. 9(a), it successfully identifies all true instances w/o any false positives and false negatives, in the top probability cluster whose probability is greater than 0.95. Note the result of DIMSUM is those points in the top probability cluster. SigGraph$^+$ identifies 6 instances with 25% FN, and Value-invariant approach identifies 48 instances with 83.3% FP. Next, we provide 67% of the pages (one truth page with 6 instances is included) to DIMSUM. As shown in Fig. 9(b), DIMSUM identifies all the 6 true instances in top probability cluster whose probability is greater than 0.80. In contrast, SigGraph$^+$ in this case identifies only 2 true instances, because for the other 4 instances, their graph patterns are not complete due to the missing pages. DIMSUM is able to survive as it aggregates enough confidence from the remaining fields. Finally, when we provide 33% pages (the truth page unfortunately is not included) to DIMSUM. It identifies nothing as shown Fig. 9(c) because all the probability is less than 0.50. SigGraph$^+$ in this case cannot find any instance as well.

2) Browser Cookies: The second case study aims to identify cookies in free pages after a browser exits. In particular, we use `w3m` and `chromium` as the benchmarks.

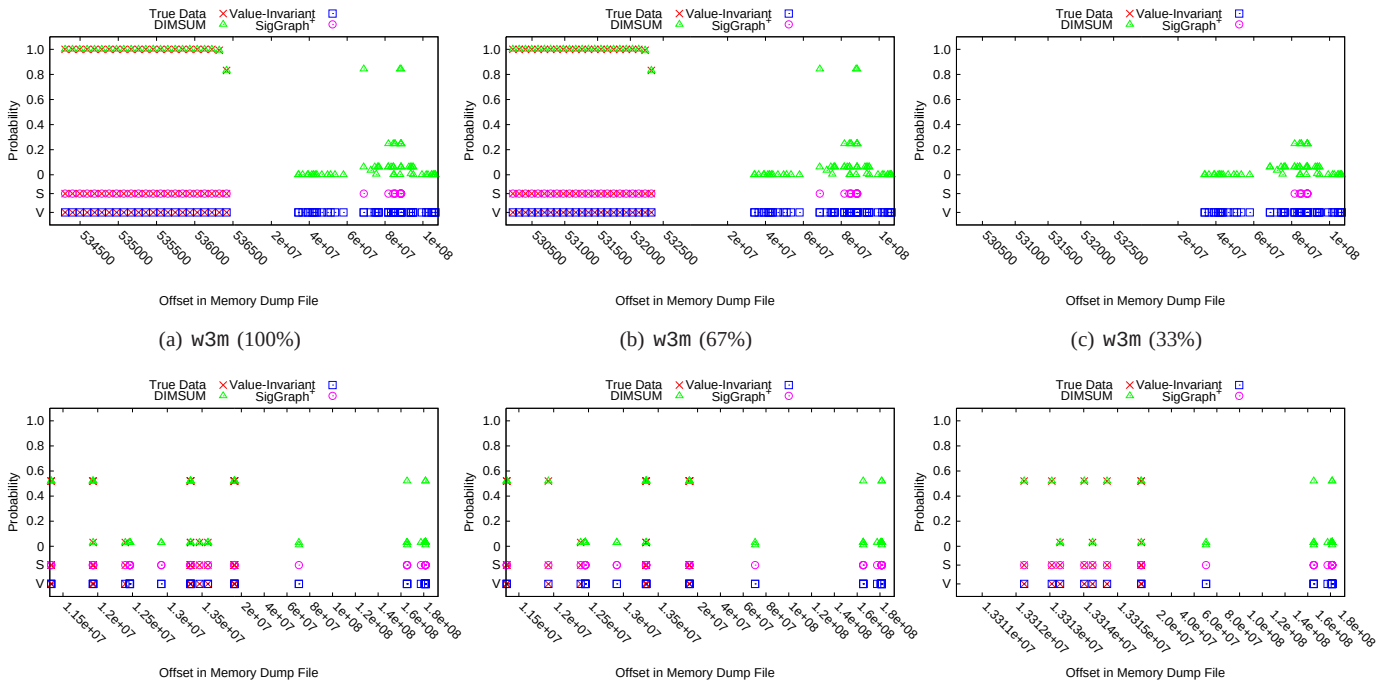
W3m The `cookie` data structure of `w3m` is organized as a link list, and each link list node has 20 fields, among which there are 15 pointers (7 pointers have `char*` type, 6 pointers have `Str*` type, 1 has an `int*` type, and 1 has a `cookie*` type), and others are 2 `char` types, 2 `int` type and 1 `time_t` (expiration time) type. The data structure `Str` has two fields: string pointer and string length.

Our snapshot has 23 cookie instances which reside in one truth page. But the data structures `Str` and normal strings pointed to by the cookie instances scatter in 137 pages. There are totally 31165 free pages. Thus, there are 138, 92 and 46 truth pages included in the 100%, 67%, and 33% settings. The results for are presented in Fig. 10(a), Fig. 10(b), and Fig. 10(c), respectively.

When all free pages are provided (100%), DIMSUM can find 22 true instances with 1 false negative. The reason is that for the last node of the link list, DIMSUM fails to collect enough support from other nodes and the probability for that node is 0.83. It hence does not belong to the top cluster of the result. With 67% pages, DIMSUM can still find these 22 true instances with 1 false negative. With 33% pages, the page with the true cookie instances is excluded and hence there is no true data. In this case, DIMSUM does not mistakenly find any false positive.

Chromium Data structure `cookie` in Chromium has 10 fields: 4 string pointers, 3 `time_t` types (indicating the creation, last access, and expiration of the cookie), and 3 `boolean` types. Unlike `w3m` which uses a link list to manage cookies, Chromium uses a general C++ `map` data structure to point to the `cookie` instances.

Our snapshot contains 25 true cookie instances which cross 10 pages, and the total number of other pages is 45298. When all truth pages are available (Fig. 10(d)), DIMSUM is able to identify 18 true instances, with 28% FP and FN. The FPs are mainly because the number of type predicates with high beliefs, determined by our primitive constraint generator, is relatively low. The resulting probability 0.5 is not high enough to distinguish true instances from others.



(a) `w3m` (100%) (b) `w3m` (67%) (c) `w3m` (33%)

(d) `chromium` (100%) (e) `chromium` (67%) (f) `chromium` (33%)

Figure 10. Effectiveness Evaluation of DIMSUM for Discovering `cookie` Data Structure.

For FNs, we examine the reason and find that some of the strings being pointed to by cookies are corrupted, and the presence of these strings is dictated by the same page constraints. In other words, such same page constraints failed, leading to low final probabilities. In comparison, SigGraph$^+$ does not have such same page constraints, and it happens to survive, resulting in no FNs in this case. With 67% and 33% pages (Fig. 10(e) and Fig. 10(f)), DIMSUM identifies 22 instances (15 are true) and 13 instances (6 true). Again, the main reason why we have nontrivial FPs and FNs for Chromium `cookie` is that the data structure has a relatively simple structure and part of its data is corrupted.

3) Email Address Book Discovery: The third case study shows how to discover the email addresses from two email client programs: `pine` and `sylpheed`.

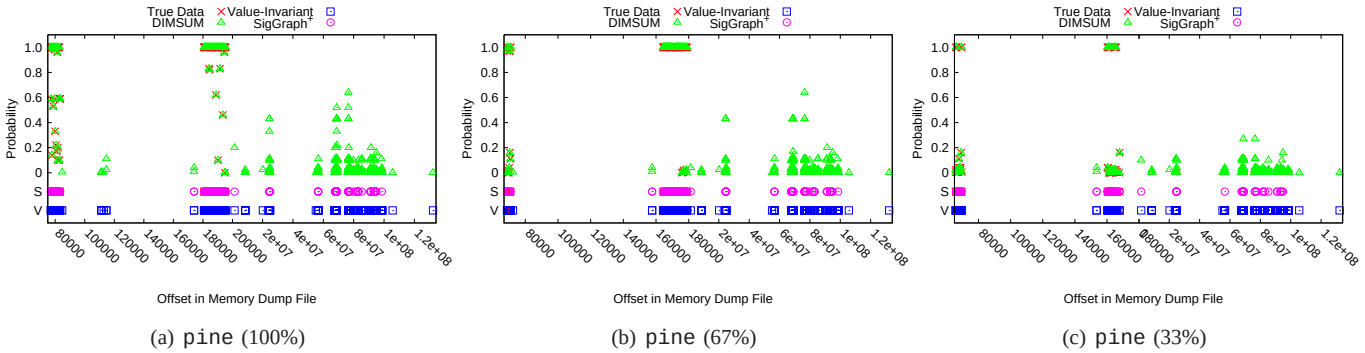
Pine The widely used email client `Pine` has its own heap object management functions, and these functions will clear data after `pine` exits. DIMSUM cannot find data if `pine` exits normally. However if `pine` crashes or gets killed, we can observe data in the free memory. As such, in this case we take a snapshot right before killing `pine`, and then analyze this snapshot with 100%, 66% and 33% true instances pages.

The address book (`adrbk_entry`) data structure for `pine` is pointed to by a number of `EntryRef` data structures that contain a limited length double linked lists each, and `adrbk_entry` has 5 string pointers. Thus, the inference graph includes not only the structure itself, but also the upper level link list and the lower level strings.

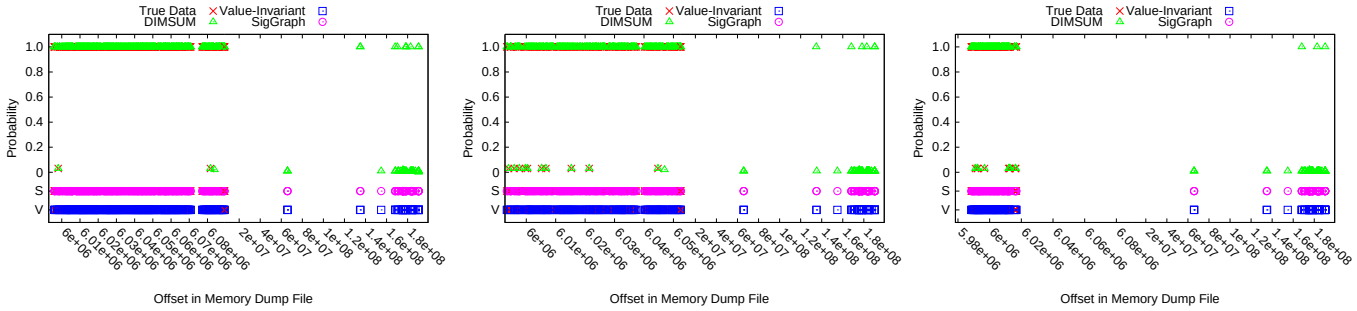
Our snapshot contains 118 true instances crossing 10 pages. The result for this case study with different truth page settings is shown in Fig. 11(a), Fig. 11(b) and Fig. 11(c), respectively. When all pages are available, DIMSUM misses 20 true instances (with FN 16.0%). In comparison, SigGraph$^+$ does not have FN at this setting. We find that the reason is that the first and the last nodes of the double link list do not have enough support, and their probabilities are a little bit lower than all other nodes in the list, and thus not included in the top cluster, although they have a higher probability than all other data. With 67% pages, DIMSUM discovered 79 true instances (w/o FP) but missed 13, and with 33% pages, DIMSUM discovers 16 true instances out of 63. When with partial true pages, the lack of support (strings and the double link list are corrupted) is the major reason for the FNs. Although SigGraph$^+$ tends to have smaller FN, it has large FP in this case.

Sylpheed Sylpheed is a GUI based email client. The address book data structure `ItemPerson` has 11 fields: 9 are pointers (6 string pointers, 2 `GList` pointers, and 1 `AddrItemObject` pointer), 1 `boolean` type, 1 `int` type, and 1 `enumeration` type.

Our snapshot has 309 instances crossing 26 pages. The experimental result for this case with different settings is shown in Fig. 11(d), Fig. 11(e), and Fig. 11(f). When all data is available (100%), DIMSUM matches 307 true instances

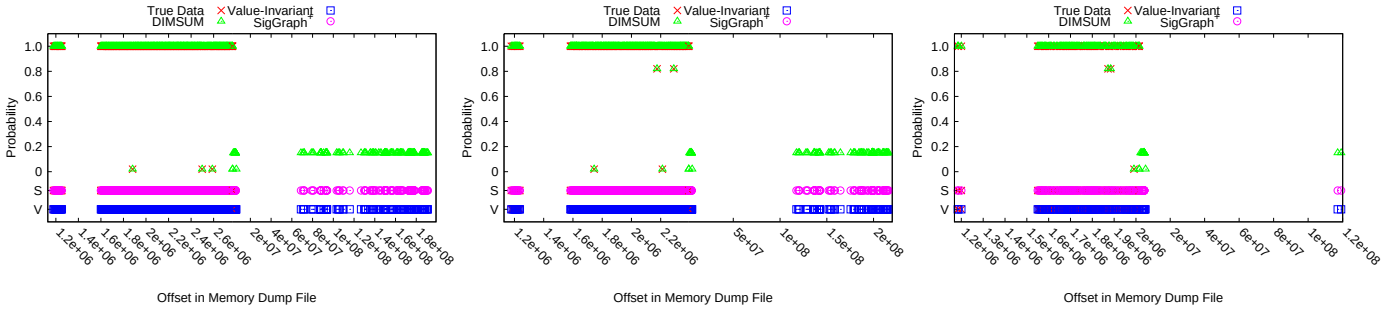


(a) `pine` (100%) (b) `pine` (67%) (c) `pine` (33%)



(d) `sylpheed` (100%) (e) `sylpheed` (67%) (f) `sylpheed` (33%)

Figure 11. Effectiveness Evaluation of DIMSUM for Discovering Email Address Book.



(a) `pidgin` (100%) (b) `pidgin` (67%) (c) `pidgin` (33%)

Figure 12. Effectiveness Evaluation of DIMSUM for Discovering Messenger Contact `buddy` List.

(missed 2), but with 16 FPs. The reason is that DIMSUM matches some false data whose pointer fields happened to be zero but satisfy our string constraint. When with 67% pages, DIMSUM identifies 205 instances (with FP 5.4% and FN 4.9%), and with 33% true pages, DIMSUM identifies 88 instances (with FP 6.8% and FN 10.9%). The reason for FP is similar to the case with 100% data. FNs are due to the missing support data.

4) Private Contact List Discovery: The final case study focuses on extracting privacy sensitive objects, that is the messenger buddy list of a universal chat client `pidgin`. In particular, each buddy of this messenger is defined in a `PurpleBuddy` data structure which has 15 fields: 13 pointer types (3 string pointers, 4 `PurpleBlistNode` pointers, 4 other `struct` type pointers, and 2 void pointers) and 2 `enumeration` types.

We run pidgin with 300 contacts, and take the snapshot right after the program exits. These 300 contacts cross 276 pages. The final result for this case study is shown in Fig. 12(a), Fig. 12(b), and Fig. 12(c). We could see we have very few FNs but no FPs in the three settings.

5) Summary on the Effectiveness: The summary of these four case studies is presented in Table III. On average, the FP for Value-invariant, SigGraph$^+$ and DIMSUM is 72.3%, 45.5%, and 7.3%, and the FN for each of them is 0.0%, 7.2%, and 12.1%, respectively. We could see DIMSUM performs significant better than SigGraph$^+$ and Value-invariant approaches in terms of the FP rate, which is

Data of Interest	Benchmark Program	#Input Pages	Truth Pages%	#True Inst.	Value-Invariant			SigGraph$^+$			DIMSUM				
					#R	FP%	FN%	#R	FP%	FN%	Factor Graph #Var	#FC	#R	FP%	FN%
Login record utmp	last 2.85	27266	100	8	48	83.3	0.0	6	0.0	25.0	507	709	8	0.0	0.0
		27265	67	6	46	87.0	0.0	2	0.0	66.7	435	609	6	0.0	0.0
		27264	33	0	40	100.0	0.0	0	0.0	0.0	405	567	0	0.0	0.0
Browser Cookies	w3m 0.5.1	31303	100	23	93	76.3	0.0	35	34.3	0.0	1874	2613	22	0.0	4.3
		31257	67	23	93	76.3	0.0	35	34.3	0.0	1874	2613	22	0.0	4.3
		31211	33	0	70	100.0	0.0	9	100.0	0.0	1260	1782	0	0.0	0.0
	chromium 8.0.552.0	45308	100	25	89	71.9	0.0	82	69.5	0.0	1068	1157	25	28.0	28.0
		45305	66	19	82	76.8	0.0	75	74.7	0.0	984	1066	22	31.8	21.1
		45302	33	9	72	87.5	0.0	63	85.7	0.0	864	936	13	53.8	33.3
Address book	pine 4.64	33186	100	118	1216	90.3	0.0	229	48.5	0.0	13056	17607	101	0.0	16.9
		33183	67	92	1174	92.2	0.0	174	50.1	6.5	11468	15594	79	0.0	14.1
		33180	33	63	1142	94.5	0.0	88	56.8	39.7	8984	12531	16	0.0	74.6
	Sylpheed 3.0.3	46504	100	309	412	25.0	0.0	412	25.0	0.0	12040	16910	307	5.0	0.6
		46496	67	204	307	33.6	0.0	307	33.6	0.0	8498	11612	205	5.4	4.9
		46487	33	92	194	52.6	0.0	194	52.6	0.0	4830	6459	88	6.8	10.9
Buddy	pidgin 2.4.1	58743	100	300	491	38.9	0.0	485	38.8	1.0	8874	12543	297	0.0	1.0
		58625	67	198	389	49.1	0.0	384	48.9	1.0	6411	8951	194	0.0	2.0
		58560	33	98	289	66.1	0.0	285	66.0	1.0	4026	5473	95	0.0	3.1

Table III
SUMMARY ON THE EFFECTIVENESS EVALUATION.

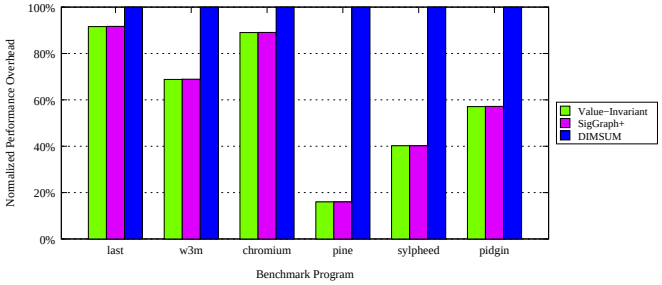


Figure 13. Performance Overhead.

particularly desirable in forensics because we don't want the user to be overwhelmed with false data. In terms of the FN rate, DIMSUM performs slightly worse than the other two but observe that the absolute rate is low anyway. Overall, we consider DIMSUM is a more balanced and more usable technique for un-mappable memory forensics.

C. Performance Overhead

In this experiment, we study the execution time cost of performing probabilistic inference in DIMSUM. The performance data is collected in a Windows Vista system with 2G memory and a 2.16Ghz CPU. The result is presented in Fig. 13. All execution times are normalized regarding DIMSUM's time. Observe that the execution time of DIMSUM is reasonable, and it takes on average 60.5% overhead for value-invariant approach, and 60.6% for SigGraph$^+$. The space overhead is decided by the factor graph size. We present the number of variables (#Var) and the number of constraints or factors (#FC) in the 12^{th} and 13^{th} columns respectively in Table III. On average, these variables and factors correspond to 161.4 MB (40358 pages) space.

VII. DISCUSSION

DIMSUM has several limitations. Firstly, if a program chooses to always zero out its data after they are used, e.g. reset all de-allocated memory, it is unlikely for DIMSUM to recover meaningful information. This is a common limitation for all forensics techniques. However, we consider that it is relatively more tedious to clean up memory than clear up other types of evidence, such as screens and files. The user has to instrument memory management functions and intercept program exit signals. In the presence of memory swapping, he/she has to make sure the pages that get swapped out are destroyed as well. Moreover, if a program crashes or gets killed, cleaning up its memory may not be easy.

Secondly, DIMSUM currently does not make use of value invariant properties such as the range of an integer field of a type T is [0,10]. Instead, it can only leverage the weaker information that it is an integer field. Value invariant properties are usually acquired from profile or domain knowledge. While it is arguable to assume profiling and domain experts in memory forensics, the success of DIMSUM without value invariant properties illustrates its unique strength. We believe DIMSUM is able to deliver better results when value invariants are leveraged.

Thirdly, our current implementation in data structure transformation demands end-users to manually write down the specification based on our grammar, in order to automatically generate constraints and then the factor graph. Part of our future work lies in making this process more automated.

VIII. RELATED WORK

Our work centers around memory forensics, a process of analyzing a memory image to explain the state of a computer system. Such process often involves discovering some data structures of interest inside the memory. The state of the art

techniques fall into two main categories: memory traversal through pointers (e.g., [1], [5], [6], [15], [19]) and signature based scanning (e.g., [3], [14], [21], [23]).

Memory traversal approach (e.g., KOP [5]) attempts to build a road map of all data structures, starting from some key addresses and traversing along points-to edges. However, such approach has to resolve pointers especially for void pointer and also cannot traverse further if pointer is corrupted. SigGraph [14] mitigates this problem by deriving a context free pointer-based signatures. Yet these techniques mostly work for live data because "dead" (i.e., freed) data cannot be reached by traversal due to missing page tables and corrupted pointers. Signature scanning directly searches memory using signatures. A classic approach is to search specific strings in memory. Other notable techniques include PTfinder [23] for linear search of Windows memory to discover process and thread structures, Volatility [21] and Memparser [3] with more capabilities of searching other types of objects.

The difference between these techniques and DIMSUM is that these techniques mainly focus on live data which is reachable, and they are either incapable of discovering unreachable data, or not sufficiently effective by generating a large number of false positives (as demonstrated in Section VI-B). DIMSUM is the first system that leverages probabilistic inference to effectively discover semantic data of interest from dead memory.

Probabilistic inference has a wide range of applications, such as face recognition (e.g., [18]), specification extraction (e.g., [12], [16]), and software debugging (e.g., [8]). 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.

IX. Conclusion

In this paper, we have demonstrated that it is possible to discover data structure instance of interest from a set of memory pages without memory mapping information. We present DIMSUM, a system that automatically extracts the data structure instance with confidence, given the data structure specification and the unmappable memory. Our experiments show that DIMSUM achieve very reasonable false negative and false positive rate.

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

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

[5] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proc. of the 16th ACM CCS*, Chicago, IL, USA, 2009.

[6] 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.

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

[8] L. Dietz, V. Dallmeier, A. Zeller, and T. Scheffer. Localizing bugs in program executions with graphical models. December 2009.

[9] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proc. of the 16th ACM CCS*, Chicago, IL, USA, 2009. ACM.

[10] B. Fabrice. Qemu, a fast and portable dynamic translator. In *Proc. of the 2005 USENIX Annual Technical Conference*, 2005.

[11] 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 *Proc. of the 17th USENIX Security Symposium*, San Jose, CA, August 2008.

[12] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *Proc. of the 7th OSDI*, Seattle, Washington, 2006.

[13] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proc. of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[14] 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 *Proc. of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[15] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proc. of the 17th Annual Network and Distributed System Security Symposium*, 2010.

[16] 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.

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

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

[19] P. Movall, W. Nelson, and S. Wetzstein. Linux physical memory analysis. In *Proc. of the FREENIX Track of the USENIX Annual Technical Conference*, Anaheim, CA, 2005.

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

[21] 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 memory. *Digital Investigation*, 3(4):197 – 210, 2006.

[22] J. Rutkowska. Klister v0.3. https://www.rootkit.com/newsread.php?newsid=51.

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

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

[25] I. Sutherland, J. Evans, T. Tryfonas, and A. Blyth. Acquiring volatile operating system data tools and techniques. *SIGOPS Operating System Review*, 42(3):65–73, 2008.

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