SIGPATH: A Memory Graph Based Approach for Program Data Introspection and Modification

David Urbina[1], Yufei Gu[1], Juan Caballero[2], Zhiqiang Lin[1]

UT Dallas[1], IMDEA Software Institute[2]
firstname.lastname@utdallas.edu, juan.caballero@imdea.org

Abstract. Examining and modifying data of interest in the memory of a target program is an important capability for security applications such as memory forensics, rootkit detection, game hacking, and virtual machine introspection. In this paper we present a novel memory graph based approach for program data introspection and modification, which does not require source code, debugging symbols, or any API in the target program. It takes as input a sequence of memory snapshots taken while the program executes, and produces a path signature, which can be used in different executions of the program to efficiently locate and traverse the in-memory data structures where the data of interest is stored. We have implemented our approach in a tool called SIGPATH. We have applied SIGPATH to game hacking, building cheats for 10 popular real-time and turn-based games, and for memory forensics, recovering from snapshots the contacts a user has stored in four IM applications including Skype and Yahoo Messenger.

Key words: program data introspection, memory graph, game hacking

1 Introduction

Many security applications require examining, and possibly modifying, data structures in the memory of a target program. These data structures store private, often sensitive, *data of interest* such as running processes in an OS, unit and resource information in online games, and credentials and contact information in Instant Messengers (IM). Such capability is crucial for memory forensics [1–4], rootkit detection [5–7], game hacking [8], reverse engineering [9–11], and virtual machine introspection (VMI) [12].

We call the process of examining in-memory data structures of a target program from an external introspector *program data introspection* (PDI). The introspector can run concurrently with the target on the same OS, e.g., a user-level debugger or a kernel module, or out-of-VM for improved isolation and higher privilege. The introspection can be done online as the program runs or offline on a snapshot of the target's memory.

The main challenges in PDI are how to efficiently locate the in-memory data structures storing the data of interest and how to traverse them to examine their data. Prior memory analysis works reuse the target's binary code or APIs to dump the data of interest (e.g., the UNIX `ps` command to list processes) [13, 14], or leverage the target's source code and debugging symbols [5, 15, 16]. However, most application-level programs do not expose external APIs to examine the data of interest and commercial off-the-self (COTS) programs rarely have source code or debugging symbols available.

In this paper we propose a novel memory-based approach for program data introspection and modification that does not require source code or debugging symbols for

the target program, or the target to expose any APIs. Our lightweight approach takes as input memory snapshots taken while the program runs and generates *path signatures* that capture how to efficiently retrieve the data of interest in memory by traversing pointers starting from the program's global variables.

Prior techniques that reverse-engineer data structures from binary programs through static (TIE [10]) or dynamic (REWARDS [9] and Howard [11]) analysis have three limitations for PDI. First, even if they recover the data structures related to the data of interest they do not address how to locate those data structures in memory. Second, they aim to recover all data structures used by a program, thus requiring very high coverage of the application's code, even when only a few data structures may be related to the data of interest. Third, they do not address recursive data structures (e.g., lists, trees) where much data of interest is stored.

More similar to our approach are techniques that compare memory snapshots such as LAIKA [17] and KARTOGRAPH [8]. However, these techniques have two limitations. First, they do not allow to reuse their results, i.e., every time the program executes they need to redo their expensive analysis. In contrast, our path signatures are created once and can then be used in many program executions. Second, they compare memory snapshots at the page level, including large chunks of dead memory. In contrast, we propose a novel technique to build a *memory graph* from a memory snapshot, structuring the live memory of a process. To the best of our knowledge ours is the first technique that can build an accurate memory graph without tracking the execution of the target program from its start and without source code or symbols, simply by using introspection on a memory snapshot. The memory graph is at the core of two techniques used to locate the data of interest in memory: *structural memory diffing*, which compares memory snapshots based on their underlying graph structure; and *fuzzing*, which modifies memory and observes its effects.

We have implemented our memory graph based approach to PDI into SIGPATH, a tool that creates path signatures to efficiently locate *and* traverse the (potentially recursive) data structures storing the data of interest. SIGPATH also provides an introspection component that takes as input a path signature and can examine and modify the data of interest while the program runs (and also from a memory snapshot). We have applied SIGPATH to two important security applications: *game hacking* and *memory forensics*.

Game hacking. The goal of game hacking is modifying the in-memory data structures of a game to cheat it into providing the player with an advantage, e.g., an immortal unit. SIGPATH's path signatures can be used to efficiently examine the game's data structures as it executes to locate the resource targeted by the cheat. Once located, it can modify its value. Our cheats need to be constructed only once and can then be used every time the game is played without further analysis. We have evaluated SIGPATH on 10 popular RTS and turn-based games, building a variety of cheats for them.

Memory forensics. The path signatures produced by SIGPATH can be used to retrieve private data from a memory snapshot of a suspect's computer. We apply SIGPATH to recover the list of contacts from a variety of IM applications a suspect may have been running when the snapshot was taken. Such a snapshot can be obtained by connecting an external memory capture device to the Firewire interface of the suspect's laptop if left unattended, even if the laptop screen was locked. We have successfully applied

SIGPATH to recover the user contacts from 4 popular IM applications including Skype and Yahoo Messenger.
In short, this paper makes the following contributions:

- We propose a novel technique to build a memory graph exclusively from a memory snapshot by introspecting the OS data structures. The memory graph is the foundation of structural memory diffing and fuzzing, two techniques for identifying the location of the data of interest in memory.
- We develop a novel memory-based path signature generation technique, which does not require source code, debugging symbols, or API information from the target program. The produced path signatures capture how to find a data structure in memory and how to traverse it (even if recursive) to examine its data.
- We implement SIGPATH, a tool that generates path signatures from memory snapshots and uses them to introspect live programs or snapshots. We evaluate SIGPATH for game hacking and memory forensics.

2 Overview & Problem Definition

SIGPATH generates path signatures that capture how to reach the data of interest by traversing pointers from the program's global variables. Our path signatures have 3 important properties: they have *high coverage*, are *stable*, and *efficient*.

First, despite their name, our path signatures capture multiple paths in memory. This is fundamental because the data of interest may be stored in an array or in a recursive data structure, e.g., a list or a tree. Here, the introspection needs to examine *all* instances of the data of interest in those data structures, e.g., all entries in a list. Second, our path signatures are stable. They are created once per program and can then be applied to any execution of that program, i.e., the paths they capture occur across executions. Third, our path signatures are efficient; they can quickly locate the data of interest by traversing a few pointers. They do not need to scan the (potentially huge) address space of the application. This makes them suitable for online introspection while the target program runs, when efficiency is key, as the program may frequently modify the introspected data structures. Of course, they can also be used offline on a snapshot of memory taken while the program was running.

Running example. To demonstrate our techniques throughout the paper we use a simplified instant messaging application (`im.exe`) that stores the contacts of the user in the data structures shown in Figure 1.

```
typedef struct _contacts {
  unsigned int num_groups;
  node_t **groups;
} contacts_t;
contacts_t *mycontacts;
```

```
typedef struct _node {
  struct _node *next;
  char *email;
} node_t;
```

Fig. 1: Data structures used by our example IM application to store the user contacts.

The application allows the user to store contacts in 16 groups (`groups` array in `contacts_t`). For each group array entry there is a linked list (`node_t`) that stores the email addresses of the users in the group. The application accesses the contact information from the `mycontacts` global variable. The data of interest are the email addresses of the user's contacts.

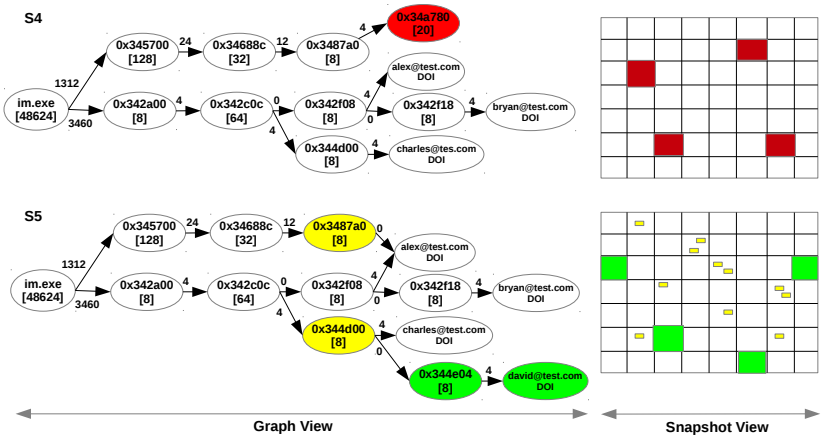

Fig. 2: On the left, simplified memory graph for two consecutive snapshots with the memory diffing results highlighted. On the right, the same memory at the page level.

2.1 The Memory Graph

The memory of a process has an inherent graph structure, which can be represented as a *memory graph*. A memory graph is a directed labeled graph $G = (V, E)$, where nodes (V) correspond to contiguous regions of *live memory*, often called buffers or objects. Nodes can represent modules loaded by the application or live heap allocations. Edges (E) represent pointers and capture the points-to relationships between live buffers when the snapshot was taken. The left side of Figure 2 shows the memory graphs for two snapshots in our running example. The path at the top of both memory graphs corresponds to data structures not shown in Figure 1.

A crucial characteristic of our approach is that it operates on a memory graph, which structures the otherwise chaotic memory of a process. The memory graph has 3 fundamental advantages over the page level representation of memory offered by a snapshot: (1) it captures only live memory; (2) it enables analyzing subgraphs of interest, e.g., only the memory reachable from the main module of the application; and (3) it captures *structural locality*, e.g., that two nodes belong to the same recursive data structure.

A snapshot of the memory of a process often comprises hundreds of MBs, of which large parts correspond to noise. This noise has two main components: dead memory and live memory largely independent of the application. Dead memory is very common in the heap. For example, LAIKA notes that only 45% of the heap is occupied by live objects, the rest being dead allocations and reserved memory not yet allocated [17]. The memory graph removes the noise introduced by dead memory. Furthermore, much live memory corresponds to external libraries, e.g., OS and graphics libraries, and the heap allocations used to store their internal data. The application requires these external libraries and their data to function but is largely independent of them. The memory graph enables further eliminating noise by removing memory only reachable by independent external libraries. The reduction in noise achieved by the memory graph is fundamental for finding the data of interest (i.e., a needle) in memory (i.e., the haystack). Our exper-

imental results show that using the memory graph reduces the size of the memory of a process that needs to be analyzed by up to 73%.

In addition, the graph representation captures structural locality, i.e., nodes in the same data structure appear close, even if they have distant starting addresses. Structural locality enables identifying recursive data structures such as lists or trees. This is fundamental because the data of interest is often stored in recursive data structures, which need to be *traversed* to extract all data of interest instances (e.g., all users in a contact list). As illustrated in Figure 2, at the page level representation it is very difficult to identify recursive data structures, thus missing the multiple instances of data of interest they may store.

Clearly, the memory graph is a very effective representation of the memory of a process, but to the best of our knowledge there is currently no solution on how to build an accurate memory graph from a memory snapshot. The main challenge when building the memory graph is to identify where the nodes are located, i.e., their start address and size. Prior works that operate on memory snapshots such as LAIKA [17] and KARTOGRAPH [8] do not offer a solution for this. In this work we propose a novel approach to build the memory graph that applies introspection on OS data structures to extract fine-grained information about the location and size of the loaded modules and the live heap allocations. We describe this process in §3.2.

2.2 Path Signatures

A path signature is a directed labeled graph $S = (V_S, E_S)$. Nodes correspond to memory buffers and can be of two types: root and normal ($V_S = V_{root} \cup V_{nor}$). Edges correspond to pointers and can be of three types: *normal*, *recursive*, and *array* ($E_S = E_{nor} \cup E_{rec} \cup E_{arr}$). A path signature has a single root node corresponding to the module (i.e., executable or DLL), where the path starts. A normal node represents a buffer in memory and is labeled with its size in bytes. Figure 3 shows the path signature for our running example where the root is the `im.exe` module and the `Normal` node corresponds to an instance of `contacts_t`.

A normal pointer corresponds to a concrete pointer in the path to the data of interest. It is a tuple (src, dst, o) where $src, dst \in V_S$ are the source and destination nodes ($src \neq dst$), and o is the offset in the src node where the pointer is stored. In Figure 3 there are 3 normal pointers (with offsets 3460,4,4) and the one with offset 3460 corresponds to the `mycontacts` global variable.

A recursive pointer is an abstraction representing that the path to the data of interest traverses a recursive data structure. During introspection SIGPATH needs to extract all instances of the data of interest in the recursive data structure, so the recursive pointer needs to be unrolled multiple times. In our running example, traversing the `next` pointer of the linked list once leads to one email, traversing it twice to a different email, and so on. A recursive pointer is a tuple $(src, dst, o, cond)$ where src, dst, o are the same as in a normal pointer and $cond$ is a boolean condition that captures when the end of the recursion

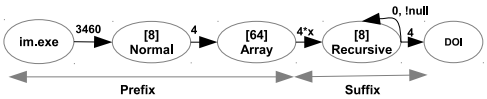

Fig. 3: Path signature for running example.

has been reached, e.g., when the recursive pointer is null. We call the source node of a recursive pointer, a recursive node. In Figure 3 the `Recursive` node corresponds to an instance of `node_t` and its recursive pointer to `next`.

An array pointer is an abstraction representing that the path to the data of interest traverses an array of pointers (or an array of a data type that contains a pointer). Again, the introspection needs to extract all instances of the data of interest reachable from the array, so all pointers in the array need to be traversed. An array pointer is a tuple $(src, dst, o, step, size)$. Iterating from o in $step$ increments until $size$ returns all offsets of the src node where a pointer to be traversed is stored. If o is zero and $size$ corresponds to the size of src we omit them and call src an array node. In Figure 3 the `Array` node corresponds to the `groups` pointer array.

A path signature comprises two parts: the *prefix* and the *suffix*. The prefix is the initial part of the path signature before any recursive or array pointer. It captures a single path in memory leading from the root to the entry point of the data structure that stores all instances of the data of interest (an array of lists in our example). The suffix captures multiple paths inside that data structure that need to be traversed to examine all instances of the data of interest. The presence of recursive and array pointers is what makes the suffix capture multiple paths. Signatures that have no suffix indicate that the program stores a single instance of the data of interest.

2.3 Approach Overview

Using SIGPATH comprises three phases: *preparation*, *signature generation*, and *introspection*. The offline preparation and signature generation phases run once to produce a stable path signature to the data of interest. The introspection phase can run many times applying the path signature to examine, and possibly modify, the data of interest in different executions of the application. In some scenarios, preparation and signature generation could be run by one entity and introspection by another, e.g., when a company sells game cheats to end-users who apply them. Next, we introduce each phase.

Preparation. The first step in our approach is to gather a sequence of memory snapshots during the execution of the application (§3.1). All snapshots come from the same execution and between two consecutive snapshots an analyst takes an application-specific action related to the data of interest. For example, in our IM application the analyst inserts new contacts, taking a snapshot after each insertion. Next, SIGPATH builds a memory graph for each snapshot, which structures the application's live memory when the snapshot was taken (§3.2). Then, SIGPATH labels which nodes in each graph store the data of interest (§3.3). To identify the data of interest it uses a combination of three techniques: *memory diffing*, *value scanning*, and *fuzzing*. Memory diffing compares consecutive snapshots guided by their graph structure to identify their differences; value scanning searches for values of the data of interest using different encodings; and fuzzing modifies values in memory while monitoring the modification's effect. Preparation is the only phase where an analyst is involved, specifically for collecting the snapshots and during fuzzing (needed for only 12% of programs).

Path signature generation. The annotated memory graphs and the snapshots are passed to the automatic path signature generation process (§4). For each memory graph it first

extracts all simple paths leading to the data of interest. Then, it identifies recursive pointers in each path and path prefixes that appear in all snapshots. Next, it locates arrays in those paths. Finally, it generalizes the paths by removing absolute addresses, and outputs stable path signatures.

Introspection. The generated path signatures can be used for examining and modifying the data of interest in different executions (or snapshots) of the application. To apply the signature, SIGPATH first identifies the base address of the root module and then traverses the signature pointers, unrolling recursive and array pointers as needed. For online introspection (and fuzzing) we have implemented two instrospectors: a user-level one that reads and modifies the target application's memory using the `ReadProcessMemory` and `WriteProcessMemory` functions, and a customized KVM monitor that introspects an application running in a VM (used for applications that check if they are monitored). For modifying the data of interest, if the data encoding is known (e.g., found through value scanning) SIGPATH can modify the data of interest to an arbitrary value. Otherwise, it can modify it to a previously observed value or fuzz it with arbitrary values until the desired effect is observed.

3 Preparation

3.1 Collecting the Snapshots

The first step in our approach is to take a number of memory snapshots while running the application. A snapshot is a dump of the content of all physical memory pages that are part of the application's address space at the time the snapshot is taken, the base virtual address of those pages, and additional metadata. SIGPATH supports two types of snapshots: Minidumps [18] produced by Microsoft's off-the-shelf tools running in parallel with the target on the same OS (e.g., WindDbg, Visual Studio, Windows Task Manager), and out-of-VM snapshots produced by the TEMU emulator [19]. The snapshots are collected at different times during the same execution. An analyst takes a snapshot before and after an action. There are positive and negative actions.

- A positive action forces the application to operate on the data of interest, leaking some information about its location in memory. For example, in our running IM example the analyst inserts a series of new contacts through the GUI taking a snapshot after each insertion. Each insertion forces the application to add the contact into its internal data structures. In a game hacking scenario where the goal is to create a unit that cannot be killed, the analyst orders one of her units to attack another of her units, forcing the game to reduce the life of the second unit.
- A negative action makes the program *not* to operate on the data of interest. The simplest negative action is to let time pass without doing anything, which helps identifying memory areas that change value independently of the data of interest.

The analyst selects one positive and one negative action for each application. It produces a sequence with an initial snapshot followed by a number of positive and negative snapshots (taken after the positive and the negative action, respectively). Obtaining both positive and negative snapshots is fundamental to structural memory diffing, detailed in

§3.3. When possible, the analyst also annotates the snapshots with the value of the data of interest being inserted or modified, e.g., the contact's email address or the unit's life. In some cases, value annotations cannot be added, e.g., when the game displays a unit's life using a bar rather than a value.

In our running example, the analyst inserts the email addresses of two contacts tagging each as group 1 and then another two tagged as group 2. It produces a sequence of 6 snapshots: one before any insertion, a positive snapshot after each insertion, and a negative snapshot a few seconds later without taking any action.

3.2 Building the Memory Graph

For each memory snapshot SIGPATH builds a memory graph $G = (V, E)$ using Algorithm 1. A node $v \in V$ corresponds to a live memory buffer and has three attributes: type, start address, and size. A node can be of 2 types: a module loaded into the application's address space (main executable or DLL) and a live heap buffer. An edge $p \in E$ is a tuple $(src, dst, o_{src}, o_{dst})$, where $src, dst \in V$ are the source and destination nodes, o_{src} is the offset in the src node where the pointer is stored, and o_{dst} is the offset into dst where the pointer points-to. For pointers to the head of the destination node we omit o_{dst}.

Algorithm 1: Memory Graph Creation

Input: A Memory Snapshot S
Output: A Memory Graph G

```
1  H ← IntrospectHeap(S);
2  M ← IntrospectModules(S);
3  X ← IntrospectStack(S);
4  V ← H ∪ M ∪ X;
5  G ← (V, ∅);
6  for each v ∈ V do
7      b ← ReadNodeBlock(S, v);
8      P ← FindPointers(V, b);
9      for each p ∈ P do
10         G.AddEdge(p);
11 return G;
```

Nodes. SIGPATH extracts the node information using introspection on the OS data structures present in the memory snapshot. Introspection is very efficient, recovering all live objects (including their size) at once from the snapshot. The alternative would be to hook the heap allocation/deallocation and module load/unload functions during program execution, tracking the size and lifetime of those objects. We have used this approach in the past and found it problematic due to: (1) being expensive and cumbersome as it has to track execution from the start (including process creation and initialization), (2) need to identify and track hundreds of Windows allocation functions built atop the Heap Manager, (3) the targeted program may contain protections against tracing (e.g., games check for debuggers and Skype rejects PIN-based tracing). These problems justify our lightweight memory introspection approach.

The introspected data structures are OS-specific and we focus on the Windows platform because that is where most proprietary programs run. SIGPATH supports both Windows 7 and Windows XP. The data structures containing the module and heap information are stored in user-level memory and can be accessed following paths starting at the Process Environment Block (PEB), whose address is included in the snapshots.

The live heap allocations can be recovered from the Windows heap management data structures. In Windows, heaps are segments of pages controlled by the Heap Manager from which the application can allocate and release chunks using OS-provided functions. Each process has a default heap provided by the OS and can create additional

heaps. Standard C/C++ functions such as `malloc` or `new` allocate memory from the CRT heap. As far as we know, no prior tool extracts the individual heap allocations of a process, so we have manually built a path signature (Fig. 4, top) to introspect the Heap Manager's data structures (based on the information on the `ntdll.dll` PDB file and external sources, e.g., [20]). SIGPATH uses this signature to automatically recover the live heap allocations from a memory snapshot. The signature captures that a process has an array of heaps (_HEAP), each containing a linked list of segments (_HEAP_SEGMENT) where each segment points to a list of the individual heap entries (_HEAP_ENTRY). Each heap entry contains the start address and size of a live heap buffer. For each heap entry SIGPATH adds a node to the memory graph.

In contrast with the heap allocations there are off-the-shelf tools that can extract the loaded modules [21, 22] but we have also built our own path signature (Fig. 4, bottom). The loaded modules can be recovered from the loader information in _PEB_LDR_DATA, which points to three lists with the loaded modules in different orderings (InLoadOrder, InMemoryOrder, and InInitializationOrder). Each list entry contains, among others, the module name, load address, and total module size. Note that for each loaded module there is a single node in the memory graph, representing the module's code and data. We could alternatively build a separate node for each module region (e.g., .data, .text) by parsing the PE header in memory. However, this is not needed as offsets from the module base are stable.

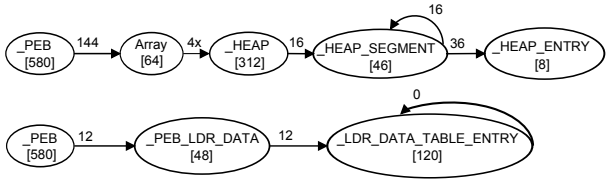

Fig. 4: Path signatures for examining the heaps and loaded modules of a process in Windows 7.

Pointers. To extract the pointers SIGPATH scans the ranges in the snapshot that correspond to live buffers. Each consecutive four bytes (8 for 64-bit) in a live buffer that forms an address pointing inside the range of a live buffer is considered a candidate pointer. This pointer detection technique is similar to the mark phase of a garbage collector and was also used by LAIKA. It can find spurious pointers (e.g., 1% of integers and 3% of strings point to the heap [17]). However, the advantage of using the memory graph is that only live memory is considered, so the probability of spurious pointers is significantly smaller. In addition, as will be shown in §4, the probability of an spurious pointer ending up in a path signature is negligible because a spurious pointer would have to appear at the same address in all snapshots and in the path to the data of interest to end up in a signature.

Reducing the graph size. Full memory graphs can contain hundreds of thousands of nodes. To reduce their size, SIGPATH includes only loaded modules shipped with the application, i.e., it excludes any libraries that are shipped with the OS. In addition, it removes any nodes not reachable from the remaining modules, e.g., those only reachable from the OS libraries. Again, as stated earlier, the memory graph we constructed can contain on average 27% of the memory in the corresponding snapshot.

3.3 Finding the Data of Interest

SIGPATH needs to identify which nodes store instances of the data of interest to extract paths leading to them. For this, it uses a combination of three techniques: *structural memory diffing*, *value scanning*, and *fuzzing*. The first two are new evolutions of previously proposed techniques for identifying data of interest in games, and our *fuzzing* is inspired by vulnerability discovery techniques.

In particular, *structural memory diffing* significantly evolves the snapshot diffing technique in KARTOGRAPH [8] and *value scanning* is used among others by the CHEAT-ENGINE [23]. Our *structural memory diffing* and *value scanning* techniques differ in that they operate on the memory graph rather than on the unstructured raw memory, which makes them substantially more accurate. For example, compared with the page level diffing in KARTOGRAPH [8], *structural memory diffing* greatly reduces the memory to diff as illustrated in Fig. 2, where the page level view (right) contains many more changes due to dead and unrelated memory. Our evaluation (§5) shows an average memory reduction of 82% on the memory graph after the application of structural memory diffing. In addition, *structural memory diffing* enables comparing the structure of the changes, which is fundamental to identify changes in the same recursive data structure.

Structural memory diffing. Given a sequence of memory graphs $G_1, \ldots, G_n$ and the corresponding snapshots $S_1, \ldots, S_n$ structural memory diffing compares each pair of consecutive memory graphs G_i, G_{i+1} extracting 3 sets: the nodes added into G_{i+1} and not present in G_i ($A_{i,i+1}$), the nodes removed from G_i and not present in G_{i+1} ($R_{i,i+1}$), and the modified nodes present in both ($M_{i,i+1}$). Fig. 2 highlights the nodes in these 3 sets after diffing the two memory graphs.

To obtain the set of changes across all snapshots, structural memory diffing computes the intersection of the sets of nodes (and byte ranges in them) modified across pairs of positive snapshots and then substracts the modifications in the negative snapshots (as they are unrelated to the data of interest):

$$M = \bigcap_{i=1}^{p-1} M^P_{i,i+1} \setminus \bigcup_{i=1}^{n-1} M^N_{i,i+1}$$

where $M^P_{i,i+1}$ and $M^N_{i,i+1}$ represent the nodes modified between snapshot i and positive or negative snapshot $i+1$, respectively.

It also computes the union of all sets of nodes added across pairs of positive snapshots minus the removed ones, and then substracts the set of nodes removed in the negative snapshots:

$$A = \bigcup_{i=1}^{p-1} A^P_{i,i+1} \setminus \bigcup_{i=1}^{p-1} R^P_{i,i+1} \setminus \bigcup_{i=1}^{n-1} R^N_{i,i+1}$$

where $A^P_{i,i+1}$ represents the set of nodes added between snapshot i and positive snapshot $i+1$, and $R^P_{i,i+1}$ and $R^N_{i,i+1}$ represent the set of nodes removed between snapshot i and positive or negative snapshot $i+1$, respectively. These two sets (M, A) are passed to the next technique as candidate locations storing data of interest.

Value scanning. If the snapshots have a value annotation, SIGPATH linearly scans the buffers output by memory diffing for the annotated value using common encodings. In our running example the first 4 snapshots are annotated respectively with:

alex@test.com, *bryan@test.com*, *charles@test.com*, and *david@test.com*. SIGPATH scans for the ASCII and Unicode versions of those email addresses. Similarly, for a unit's life it uses encodings such as short, integer, long, float, and double.

Note that we only use value scanning during preparation, as scanning memory at introspection while the program runs is too slow and precisely one of the reasons to generate path signatures. Also, value scanning is applied after structural memory diffing has significantly reduced the number of candidates. This is important because some values may not be discriminating enough and would match many locations otherwise. Value scanning is a simple yet surprisingly effective technique, and if it finds a value it also identifies its encoding. However, it does not work when the value is very common (e.g., zero); when the data uses non-standard encoding; is obfuscated or encrypted; or when the value to search is unknown.

Fuzzing. If multiple candidate nodes are still left the analyst uses fuzzing. In fuzzing, the analyst modifies the value of memory locations and monitors the effect of the modification. If the modification produces a visually observable result such as modifying the unit's life or changing the name of a contact on the screen, then the node stores the data of interest. Fuzzing is an online technique, similar to the process of modifying the data of interest in game hacking.

4 Path Signature Generation

The input to the signature generation is a sequence of pairs $\langle S_i, G_i \rangle$ where G_i is the memory graph for snapshot S_i, annotated with the nodes storing the data of interest. The output is a set of stable path signatures. Algorithm 2 describes the signature generation. For each memory graph G_i, SIGPATH first extracts the set of simple paths rooted at one of the loaded modules and leading to any of the nodes labeled as data of interest (line 4). Simple paths do not contain repeating vertices (i.e., no loops) and for each data of interest node SIGPATH may extract multiple simple paths, possibly rooted at different nodes. Fig. 5 shows the 5 paths extracted from the 4 positive snapshots (S2–S5) in our running example.

Algorithm 2: Path Signature Generation

Input: A sequence of pairs $\langle S_i, G_i \rangle$, and the data of interest d
Output: A set of stable path signatures $SIGS$ for d

1 $SIGS \leftarrow \emptyset$;
2 $NSP \leftarrow \emptyset$;
3 **for each** $G_i \in \langle S_i, G_i \rangle$ **do**
4 $\quad SP_i \leftarrow ExtractSimplePaths(G_i, d)$;
5 $\quad NSP \leftarrow NSP \cup FindRecursivePointers(SP_i)$;
6 $CP \leftarrow FindSetsOfPathsWithCommonPrefix(NSP)$;
7 **for each** $pathSet \in CP$ **do**
8 $\quad C \leftarrow ChunkPaths(pathSet)$;
9 $\quad$ **if** $EquivalentChunks(C)$ **then**
10 $\quad\quad A \leftarrow FindArrays(C)$;
11 $\quad\quad sig \leftarrow MergeChunks(C, A)$;
12 $\quad\quad SIGS \leftarrow SIGS \cup sig$;
13 $SIGS \leftarrow GeneralizePathSignatures(SIGS)$;
14 **return** $SIGS$;

Next, SIGPATH searches for recursive pointers in each of the paths (line 5). A recursive pointer is a pointer that points-to a node of the same type as the node holding the pointer. If a path contains two consecutive nodes of the same type, then the pointer linking them is recursive. As we do not know the type of each node, we approximate it using its size and internal structure. In particular, for each pair of consecutive nodes of

the same size, we compare the offsets of all the pointers stored in each node, including those not in the path and taking special care of null pointers, as well as the offset of any ASCII or Unicode string they may store. If the offsets are identical, we consider the nodes to be of the same type and mark the pointer linking them as recursive.

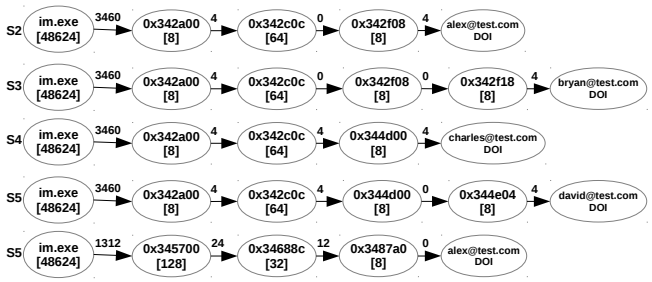

Fig. 5: Paths from 4 snapshots of running example.

Then, SIGPATH searches for stable prefixes, which are path prefixes that appear in at least one path from each memory graph (line 6). Such prefixes capture structural proximity, i.e., data in the same data structure. To identify stable prefixes SIGPATH compares the paths node by node, considering two nodes from different paths equivalent if the source offset of the pointer leading to them, their address, and their size are identical. Shared prefixes stop at recursive pointers. Once the stable prefixes are identified, any paths that do not contain a stable prefix are removed as they cannot lead to a stable path signature. If no stable prefix is found, the process stops. However, this situation is unlikely and does not happen in our experiments. Fig. 6 shows the paths in our running example after identifying recursive pointers and stable prefixes.

Stable prefixes that reach the data of interest correspond already to stable path signatures and can proceed to generalization. For the remaining paths, SIGPATH splits them into chunks (line 8), ending a chunk at a recursive node (dashed vertical lines in Fig. 6). Then, it compares the shape of each chunk across the paths. If the nodes in the chunk have the same address, are data of interest nodes, or have the same size and internal structure, they are considered equivalent. In Fig. 6 all four nodes in chunks 1 and 2 are equivalent. If the chunks are not equivalent, the shared prefix cannot generate a stable signature, so the paths containing them are removed (line 9).

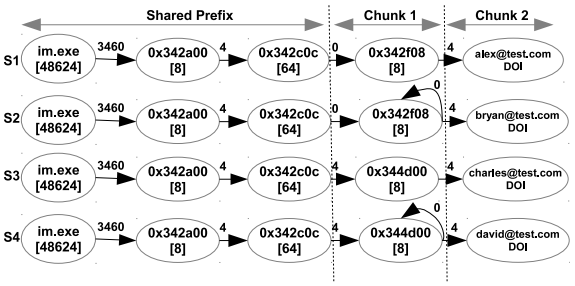

Fig. 6: Recursive pointers and stable prefix in paths from running example.

For each chunk where the nodes are equivalent, SIGPATH looks for arrays (line 10). If the offset of the pointers leading to the chunk are different across snapshots there may be an array that is being exited through different array elements. This is the case with the last node in the shared prefix in Fig. 6, which leads to chunk 1 through offset 0 and 4 in different snapshots. Here, SIGPATH computes the greatest common denominator (gcd) of the offsets of the pointers leading to the chunk. If the gcd is larger or equal to the

word size, the node is flagged as an array with $step = gcd$. The array size is determined by the largest offset leading to the chunk plus the $step$, accounting for NULL entries and being limited by the heap object size.

At this point, only stable paths that reach the data of interest in each of the snapshots remain and their chunks and arrays can be combined into a path signature (line 11). Finally, the path signatures are generalized by removing the nodes' memory addresses, since the same node on different executions will have different addresses (line 13). Fig. 3 shows the resulting stable path signature for our running example.

5 Evaluation

We have implemented the memory graph construction, finding the data of interest, and path signature generation components of SIGPATH using 3,400 lines of Python code (excluding external libraries). In addition, the user-level introspector comprises 1,500 lines of C++ and the KVM introspector with another 1,800 lines of C. In this section we present our evaluation results.

Experiment setup. Table 1 presents the applications used to evaluate SIGPATH. For each application, it details the different data of interest for which we use SIGPATH to produce a path signature. Applications are grouped into 3 categories: instant messengers (IM), real-time games (RTS), and board games (BG). The goal with IM applications is to retrieve from a given memory snapshot the user contacts the application stores (email address or contact identifier). In RTS games the goal is to modify the game's state at runtime to gain advantage, i.e., to cheat. The cheater plays against a variable number of adversaries controlled by remote players (or by the computer), and his goal is to (unfairly) beat them by obtaining an unlimited number of resources and units that heal themselves. Finally, in board games, the cheater plays against the computer and his goal is to set high scores to impress his friends.

Table 1: Applications used in our evaluation.

Type	Application	Version	Data of Interest
IM	Yahoo Msg	11.5	Email
	Skype	6.6.0.106	Contact ID
	Pidgin	2.10.7	Email
	Miranda	0.10	Contact ID
RTS	SimCity 4	–	Money
	StarCraft	1.15	Mineral, Gas, CC life, SCV life
	Age of Empires II	0.14	Gold, Wood, Iron, Food
	WarCraft III	1.24	Gold, Wood
	Dune 2000	1.02	Money
	Emperor	1.04	Money
	Warlords Battlecry III	1.03	Gold, Hero experience
	C&C Generals	1.80	Money
BG	Spider Solitaire	5.10	Cards, Points
	Monopoly	1.00	Money

All applications are 32-bit and run on both Windows 7 and Windows XP SP3 Virtual Machines. Experiments are performed on an host Intel Xeon with 16 cores 2.4GHz, with 48GB RAM, running Red Hat Enterprise 64-bit with kernel 2.6.32.

Collecting the snapshots. For each application and data of interest we select a positive action and take snapshots after performing the action. For IM applications the positive action is inserting a new contact. For RTS games and resources (i.e., money, mineral, gas, gold, wood, iron, food) it is to put a unit to collect the resource, taking a snapshot when the resource counter increases. For StarCraft and the life of a unit (CC life, SCV

Table 2: Evaluation results for the preparation and signature generation phases.

		Snapshot Collection		Memory Graph Generation					Finding the Data of Interest				Signature Generation		
Application	DOI	#Snaps.	Avg. Size (MB)	Avg. #Nodes	Avg. #Ptr	Avg. Size (MB)	Avg. Filtered Size (MB)	Avg. Ratio (%)	Memory Diffing (MB)	Value Scanning (bytes)	Fuzzing (bytes)	Encoding	#Nodes	#Paths	#Sig.
Yahoo Msg	Email	6	93	8,422	14,837	20	19	20	2	16	-	ASCII	2	6	1
Skype	Contact ID	6	102	3,761	12,618	70	63	61	8	20	-	ASCII	7	10	1
Pidgin	Email	6	52	6,305	41,981	5.6	5.5	10	1	32	-	ASCII	7	123	2
Miranda	Contact ID	6	34	671	5,790	6.2	6	17	2	16	-	ASCII	4	11	1
SimCity 4	Money	6	269	32,643	128,637	102	97	36	13	4	-	Integer	1	1	1
StarCraft	Minerals	10	45	1,069	1,672	5.5	5.2	12	1.1	4	-	Integer	1	1	1
	Gas	10	45	1,069	1,672	5.5	5.2	12	1.1	4	-	Integer	1	1	1
	CC life	10	45	1,069	1,672	5.5	5.2	12	1.1	2	-	Short	1	1	1
	SCV life	10	45	1,069	1,672	5.5	5.2	12	1.1	2	-	Short	1	1	1
Age of Empires II	Gold	6	157	488	1,709	85	79	50	5	4	-	Float	1	1	1
	Food	6	157	488	1,709	85	79	50	5	4	-	Float	1	1	1
	Wood	6	157	488	1,709	85	79	50	5	4	-	Float	1	1	1
	Iron	6	157	488	1,709	85	79	50	5	4	-	Float	1	1	1
WarCraft III	Gold	6	269	4,419	12,649	27	24	8	9	4	-	Integer	1	2	1
	Wood	6	269	4,419	12,649	27	24	8	9	4	-	Integer	1	2	1
Dune 2000	Money	6	40	842	1,575	10	10	4	2	4	-	Integer	1	1	1
Emperor	Money	10	114	37,129	10,232	70	65	57	7	102	4	Integer	1	4	1
Warlords Battlecry III	Gold	6	218	13,242	5,784	42	41	18	1	2	-	Short	1	1	1
	Hero XP	6	218	13,242	5,784	42	41	18	1	2	-	Short	1	1	1
C&C Generals	Money	10	339	2,537	140,157	68	56	17	10	4	-	Integer	1	2	1
Spider Solitaire	Cards	6	26	372	819	3	3	12	1	-	4	(Integer)	18	18	1
	Points	6	26	372	819	3	3	12	1	4	-	Integer	1	1	1
Monopoly	Money	6	114	4,628	55,389	24	19	17	8	2	-	Short	1	2	1

life) it is to order one unit to attack the target unit (CC, SCV) taking a snapshot whenever the target's life decreases. For Warlords Battlecry III and the Hero experience it is to use the Hero unit to kill other units, taking a snapshot after each killing, which increases the Hero's experience. For Spider (both cards and points) it is to move a card from one stack of cards to another and for Monopoly to buy some new property, which reduces the player's money. By default we collect 6 snapshots for each application. We take an initial snapshot, then apply the positive action 4 times, and finally collect a final snapshot after 5 minutes of inactivity (negative action). When finding the data of interest, if SIGPATH cannot locate it (i.e., too many candidate locations are left), we redo this process doubling the number of positive actions.

The first part of Table 2 summarizes the snapshot collection. It presents the number of snapshots taken for each game and the average size in Megabytes of those snap-

shots. For all games except StarCraft, Emperor, and C&C Generals the first batch of 6 snapshots was enough to identify the data of interest. These three games exhibit a large number of memory changes between snapshots, which makes it more difficult to pinpoint the data of interest. Taking a second batch of snapshots was enough to locate the data of interest. The snapshots are in Windows Minidump format and contain all user-level memory pages (0–0x7fffffff). The largest snapshots correspond to RTS games and all applications increase their memory usage as time passes.

Generating the memory graphs. The next part of Table 2 shows the average number of nodes and pointers in the memory graphs generated from the snapshots. These numbers are obtained after removing all nodes unreachable from the modules shipped with the program's executable, i.e., those only reachable from Windows libraries. The numbers show that memory graphs can be large comprising up to tens of thousands of nodes and over a hundred thousand pointers. However, their total size is still a small fraction of the snapshot size, as an important fraction of the pages in the snapshot corresponds to dead memory and OS data structures. Additionally, we also present the average size of the graph before and after removing unreachable nodes. Finally, we show the ratio between the average graph size and the average snapshot size, representing the fraction of the original process memory that needs to be examined. The results show that using the memory graph translates in an average 73% reduction of the memory space.

Finding the data of interest. The following part of Table 2 captures the combination of techniques needed to identify the data of interest, its encoding, and the total number of nodes flagged storing data of interest. We depicted the average size of memory space where the data of interest must be searched for after the application of each technique. A dash sign represents that the technique was not needed.

In 22 of 23 cases the combination of structural memory diffing and value scanning located the data of interest. This combination is very powerful as structural memory diffing first reduces the set of candidate locations up to 82% on average of the memory graph, and then value scanning pinpoints the data of interest. Without memory diffing, value scanning flagged too many candidate locations. For the money in Emperor, we needed to apply fuzzing as two candidate nodes remained after memory diffing and value scanning. For the cards in Spider, value scanning does not apply because we move cards rather than inserting data. In this case, a combination of structural memory diffing and fuzzing identifies the data of interest.

Signature generation. The last part of Table 2 shows the number of paths passed to the signature generation and the final number of path signatures produced. For cases where multiple paths are input to the signature generation, the common prefix extraction identifies which of those paths are stable and discards the rest. For all applications except Pidgin, there is only one stable path and thus have a single path signature. We find that Pidgin stores two different copies of the contact's email address. Both of them are stored in the same contacts data structure but their paths differ on the final steps. Both paths are stable producing two different signatures, both of which can be used to identify the contact emails during introspection.

For RTS games and Monopoly, the path signatures have a single node indicating that those games store resources, unit life, and unit experience in global variables, likely for performance reasons. One could naively think that such signatures do not contain im-

Table 3: Path signatures produced by SIGPATH.

Application	DOI	Segment	Example Signature
Yahoo Msg	Email	Heap	yahoo.exe[2394323]→ A(32)[4]→ B(46)[*4][32]→ C(12)[4]→ D[0]"Email"
Skype	Contact ID	Heap	skype.exe[7930844]→ A(64)[60]→ B(2062)[136]→ C(686)[16]→ D(488)[244] → E(553)[360] → F(256)[8*x]→ G[0]"Contact ID"
Pidgin	Email	Heap	libpurple.dll[495776]→ A(12)[0]→ B(48)[16]→ C(56)[*8][48]→ D(64)[32] → E[0]"Email"
		Heap	libpurple.dll[495776]→ A(12)[0]→ B(48)[16]→ C(56)[*8][48]→ D(64)[56] → E(40)[32]→ F[0]"Email"
Miranda	Contacts ID	Heap	yahoo.dll[147340]→ A(12)→ B(92)[32]→ C(12)[*0][8]→ D(28)[4] → E[0]"Contact ID"
SimCity 4	Money	Global	SimCity 4.exe[25674647]"Money"
StarCraft	Mineral	Global	StarCraft.exe[1569012]"Mineral"
	Gas	Global	StarCraft.exe[1569060]"Gas"
	CC Life	Global	StarCraft.exe[2241505] "Command Center Life"
	SCV life	Global	StarCraft.exe[2241231]"SCV life"
Age of Empires II	Gold	Global	empires2.exe[4564562]"Gold"
	Wood	Global	empires2.exe[4564566]"Gas"
	Food	Global	empires2.exe[4564570]"Food"
	Iron	Global	empires2.exe[4564574]"Iron"
WarCraft III	Gold	Global	war.exe[3022425]"Gold"
	Wood	Global	war.exe[3022445]"Wood"
Dune 2000	Money	Global	dune2000.dat[3901300]"Money"
Emperor	Money	Global	game.dat[3530433]"Money"
Warlords Battlecry III	Gold	Global	Battlecry III.exe[3400232]"'Gold"
	Hero XP	Global	Battlecry III.exe[2374828]"Hero XP"
C&C Generals	Money	Global	game.dat[55456432]"Money"
Spider Solitaire	Card	Heap	spider.exe[73744]→ A(120)[4*x]→ B(4)[0]→ C(12)[*8]"Card"
	Points	Global	spider.exe[77664]"Points"
Monopoly	Money	Global	Monopoly.exe[3230202]"Money"

portant information, but this is hardly so. They capture at which offset and in which module the global variable is stored. This enables to quickly introspect the application without having to scan all modules for a value that is likely to appear many times. During preparation, our approach combines memory diffing, value scanning, and fuzzing to accurately pinpoint the location of the data of interest, so that there is no need to check during online introspection which candidate is the right location. The root of the path signatures is not always the main module of the application, e.g., *libpurple.dll* in Pidgin. A special case happens with games where the main executable starts a separate process in charge of the game, such as in Emperor and C&C Generals. Table 3 shows the path signatures in a path-like representation.

Performance evaluation. We have measured the execution time and memory usage of SIGPATH. On average it takes an analyst 25 minutes to collect the 6 snapshots, the largest being 45 minutes for Warcraft III. After that, SIGPATH takes on average 10 minutes to produce the path signatures. The slowest generation is 22 minutes for Pidgin due to the large number of candidate paths. Overall, it takes 35–60 minutes to create the signatures. Signature generation is memory consuming because it requires operating on large memory graphs and snapshots. The largest memory consumption was C&C Generals with close to 4GB. We plan to add optimizations to minimize the number of memory graphs and raw pages from snapshots to be simultaneously loaded in memory.

Application: Game hacking. For each tested game in Table 2 and data of interest pair we create a cheat and use SIGPATH to identify the location of the data of interest in memory, modifying it with the cheat's value. During preparation we identified the encoding for each data of interest, so here we only need to select an appropriate value. For RTS games we select a large number corresponding to a ridiculously high amount of resources, unit life, or unit experience. For board games we select a very large score and make sure that the game stores it in its high score list. The cheats work flawlessly. As an example, one cheat for StarCraft increases the Command Center unit life to 9,999 points, even if the maximum life of that unit type is 1,500 points. After the cheat the game continues operating correctly and the unit's life is accepted to be 9,999. Setting the life to zero is even a better cheat as it makes the unit immortal.

Application: Memory forensics. We evaluated whether our path signatures are able to recover all email contacts stored by the IM programs in Table 2, given a snapshot taken from a suspect's computer. We run each IM application and insert 3 new contacts (different from the ones inserted when collecting the snapshots during preparation) taking a single snapshot for each application after all insertions. Then, we use the introspection component of SIGPATH to apply the signature on the snapshot. SIGPATH successfully identifies and dumps the email addresses of all 3 contacts in all 4 IM applications. This demonstrates that the generated path signatures can examine all instances of the data of interest stored in a data structure. It also demonstrates that they can be used to examine a different execution from the one used to generate them.

6 Limitations and Future Work

Obfuscation. Our approach works with many common code protections such as those used by Skype [24]. For data obfuscation, if the data of interest is encrypted SIGPATH can still find it using structural memory diffing and fuzzing, and can modify it by replaying a previously seen (encrypted) value. Other obfuscation techniques such as data structure randomization (DSR) [25] target data analysis techniques. SIGPATH cannot handle DSR, but DSR requires expensive code refactoring to existing programs.

Correlating actions and effects. Our memory diffing technique requires positive actions chosen by the analyst to affect the data of interest. And, our fuzzing technique requires modifications of the data of interest to be observable by the analyst. Such direct relationships between analyst actions and effects on the data of interest was easy to find in our applications. However, other scenarios may prove more difficult.

Complex code idioms. Some complex code idioms are unsupported or only supported partially by SIGPATH. For example, while SIGPATH handles most unions, a subtle case is a union of two types, both with a pointer at the same offset and only one those pointers leading to the data-of-interest, which produces an unstable path. Another issue are tail-accumulator arrays; while our array detection technique identifies the array at the end of the node, full support would require handling variable-length nodes. Also problematic are data structures whose traversal requires computation, e.g., masking the bottom 7 bits of a pointer field used as a reference count. Finally, custom allocators hide the proper structure of memory, but can be identified by recent techniques [26].

3D support. VirtualBox, QEMU, and XEN do not support 3D graphics, required by some recent games like StarCraft2. We are developing a VMWare introspector for these.

7 Related Work

Game hacking. Hoglund and McGraw [27] describe common game cheating techniques such as memory editing, code injection, and network traffic forgery. CHEAT-ENGINE [23] is an open source engine that uses value scanning to identify the data of interest while the game runs. As acknowledged by the author, such approach does not work with highly dynamic online games. KARTOGRAPH [8] is a state-of-the-art tool for hacking games. It differs from SIGPATH in two key properties. First, SIGPATH produces path signatures, which can be reused to cheat the game in many runs without redoing the analysis. In addition, SIGPATH operates on memory graphs while KARTOGRAPH operates at the page level. The memory graph removes dead memory and unrelated data, and enables structural memory diffing, which improves accuracy over the page level diffing in KARTOGRAPH.

Data structure reverse engineering. Prior works recover data structures using program analysis without source code or debugging symbols, e.g., REWARDS [9], HOWARD [11], and TIE [10]. In contrast, SIGPATH employs a lightweight memory-based approach that locates the data of interest and generates a path signature to it, avoiding the need to recover all data structures. Most similar is LAIKA [17], which applies machine learning to identify data structures in a memory snapshot, clusters those of the same type, and computes the similarity of two snapshots. SIGPATH differs in that it produces path signatures that identify the exact location of the data of interest in memory and that its structural memory diffing operates on a memory graph rather than at the page level.

Memory forensics. Prior work identifies data structures in memory by traversing pointers starting from program (kernel) global variables and following the points-to relationships to reach instances of the data structure. KOP [5], MAS [16], FATKIT [1] and VOLATILITY [3] all use such technique. While SIGPATH also leverages this approach the substantial difference is that these works require access to the target's source code or its data structure definitions in symbol files. They also often involve sophisticated points-to analysis (e.g., for `void` pointers), whereas SIGPATH simply requires memory snapshots. Instead of object traversal, other works use data structure signatures to *scan* for instances of data structures in memory, such as PTFINDER [2], Robust-Signatures [6], and SIGGRAPH [7]. SIGPATH differs in that it produces path signatures that do not require to scan memory and also in that it does not require access to the type definitions in the source code.

Virtual machine introspection (VMI). LIVEWIRE [12] demonstrated the concept of VMI and there is a significant amount of works that improve VMI for better practicality, automation, and wider applications such as VMWATCHER [15], SBCFI [28],VIRTUOSO [13] and VMST [14]. The difference with SIGPATH is that these systems focus on kernel level introspection, whereas SIGPATH focuses on user level data introspection. Recently, TAPPAN ZEE BRIDGE [29] finds hooking points at which to interpose for active monitoring. Our work differs on being snapshot-based and producing path signatures.

8 Conclusion

In this paper we have presented a novel memory graph based approach for program data introspection and modification that generates path signatures directly from memory snapshots, without requiring source code, debugging symbols, or APIs in the target program. To this end, we have developed a number of new techniques including a technique for generating a memory graph directly from a snapshot without tracking the program execution; a structural memory diffing technique to compare the graph structure of two snapshots; and a signature generation technique that produces path signatures that capture how to efficiently locate and traverse the in-memory data structures where the data of interest is stored. We have implemented our techniques into a tool called SIGPATH and have evaluated SIGPATH for hacking popular games and for recovering contact information from IM applications.

9 Acknowledgements

The authors thank the anonymous reviewers for their feedback. This material is based upon work supported by The Air Force Office of Scientific Research under Award No. FA-9550-12-1-0077. This research was also partially supported by the Spanish Government through Grant TIN2012-39391-C04-01 and a Juan de la Cierva Fellowship for Juan Caballero. All opinions, findings and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

References

1. Petroni, N.L., Jr., Walters, A., Fraser, T., Arbaugh, W.A.: Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. Digital Investigation **3**(4) (2006) 197 – 210
2. Schuster, A.: Searching for processes and threads in Microsoft Windows memory dumps. Digital Investigation **3**(Supplement-1) (2006) 10–16
3. Walters, A.: The volatility framework: Volatile memory artifact extraction utility framework `https://www.volatilesystems.com/default/volatility`.
4. Lin, Z., Rhee, J., Wu, C., Zhang, X., Xu, D.: Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In: Proceedings of Network and Distributed System Security Symposium, San Diego, CA (February 2012)
5. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping Kernel Objects to Enable Systematic Integrity Checking. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, Chicago, IL (November 2009)
6. Dolan-Gavitt, B., Srivastava, A., Traynor, P., Giffin, J.: Robust Signatures for Kernel Data Structures. In: Proceedings of the 16th ACM conference on Computer and Communications Security, Chicago, IL (November 2009)
7. Lin, Z., Rhee, J., Zhang, X., Xu, D., Jiang, X.: 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, San Diego, CA (February 2011)
8. Bursztein, E., Hamburg, M., Lagarenn, J., Boneh, D.: OpenConflict: Preventing Real Time Map Hacks in Online Games. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA (May 2011)

9. Lin, Z., Zhang, X., Xu, D.: Automatic Reverse Engineering of Data Structures from Binary Execution. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium, San Diego, CA (February 2010)
10. Lee, J., Avgerinos, T., Brumley, D.: TIE: Principled Reverse Engineering of Types in Binary Programs. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium, San Diego, CA (February 2011)
11. Slowinska, A., Stancescu, T., Bos, H.: Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium, San Diego, CA (February 2011)
12. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the 10th Annual Network and Distributed Systems Security Symposium, San Diego, CA (February 2003)
13. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA (May 2011)
14. Fu, Y., Lin, Z.: Space Traveling across VM: Automatically Bridging the Semantic-Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In: Proceedings of the IEEE Symposium on Security and Privacy, San Francisco, CA (May 2012)
15. Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection Through VMM-Based Out-of-the-Box Semantic View Reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA (November 2007)
16. Cui, W., Peinado, M., Xu, Z., Chan, E.: Tracking Rootkit Footprints with a Practical Memory Analysis System. In: Proceedings of the USENIX Security Symposium. (August 2012)
17. Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging for Data Structures. In: Proceedings of the 8th Symposium on Operating System Design and Implementation, San Diego, CA (December 2008)
18. Microsoft: Minidump definitions `http://msdn.microsoft.com/en-us/library/windows/desktop/ms680378.aspx`.
19. Yin, H., Song, D.: TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, CA (January 2010)
20. McDonald, J., Valasek, C.: Practical windows xp/2003 heap exploitation (2009)
21. Russinovich, M., Cogswell, B.: Vmmap `http://technet.microsoft.com/en-us/sysinternals/dd535533.asp`.
22. Russinovich, M.: Process explorer `http://technet.microsoft.com/en-us/sysinternals/bb896653`.
23. Team, C.E.: Cheat engine `http://www.cheatengine.org/`.
24. Biondi, P., Desclaux, F.: Silver Needle in the Skype. In: BlackHat Europe. (March 2006)
25. Lin, Z., Riley, R.D., Xu, D.: Polymorphing Software by Randomizing Data Structure Layout. In: Proceedings of the 6th SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment, Milan, Italy (July 2009)
26. Chen, X., Slowinska, A., Bos, H.: Who Allocated my Memory? Detecting Custom Memory Allocators in C Binaries. In: Working Conference on Reverse Engineering. (October 2013)
27. Hoglund, G., McGraw, G.: Exploiting Online Games: Cheating Massively Distributed Systems. First edn. Addison-Wesley Professional (2007)
28. Petroni, Jr., N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA (October 2007)
29. Dolan-Gavitt, B., Leek, T., Hodosh, J., Lee, W.: Tappan zee (north) bridge: Mining memory accesses for introspection. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security. (Nov 2013)