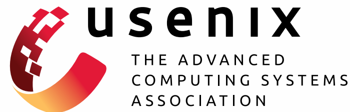


DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse

Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu, ***Purdue University***

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/saltaformaggio

This paper is included in the Proceedings of the 23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

Open access to the Proceedings of the 23rd USENIX Security Symposium is sponsored by USENIX

DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse

Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, Dongyan Xu
Department of Computer Science and CERIAS
Purdue University, West Lafayette, IN 47907
{*bsaltafo, gu16, xyzhang, dxu*}*@cs.purdue.edu*

Abstract

State-of-the-art memory forensics involves signature-based scanning of memory images to uncover data structure instances of interest to investigators. A largely unaddressed challenge is that investigators may not be able to *interpret the content of data structure fields*, even with a deep understanding of the data structure's syntax and semantics. This is very common for data structures with application-specific encoding, such as those representing images, figures, passwords, and formatted file contents. For example, an investigator may know that a `buffer` field is holding a photo image, but still cannot display (and hence understand) the image. We call this the *data structure content reverse engineering* challenge. In this paper, we present DSCRETE, a system that enables automatic interpretation and rendering of in-memory data structure contents. DSCRETE is based on the observation that the application in which a data structure is defined usually contains interpretation and rendering logic to generate human-understandable output for that data structure. Hence DSCRETE aims to *identify and reuse* such logic in the program's *binary* and create a "scanner+renderer" tool for scanning and rendering instances of the data structure in a memory image. Different from signature-based approaches, DSCRETE avoids reverse engineering data structure signatures. Our evaluation with a wide range of real-world application binaries shows that DSCRETE is able to recover a variety of application data — e.g., images, figures, screenshots, user accounts, and formatted files and messages — with high accuracy. The raw contents of such data would otherwise be unfathomable to human investigators.

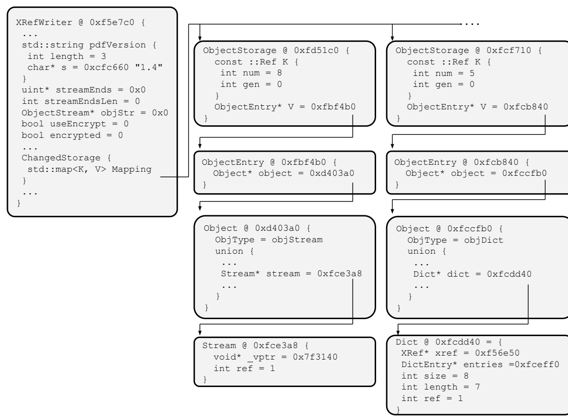
1 Introduction

Traditionally, digital investigations have aimed to recover evidence of a cyber-crime or perform incident response via analysis of non-volatile storage. This routine involves powering down a workstation, preserving images of any storage devices (e.g., hard disks, thumb drive, etc.), and later analyzing those images to recover evidentiary files. However, this procedure results in a significant loss of *live evidence* stored in the system's RAM — executing processes, open network connections, volatile IPC data, and OS and application data structures.

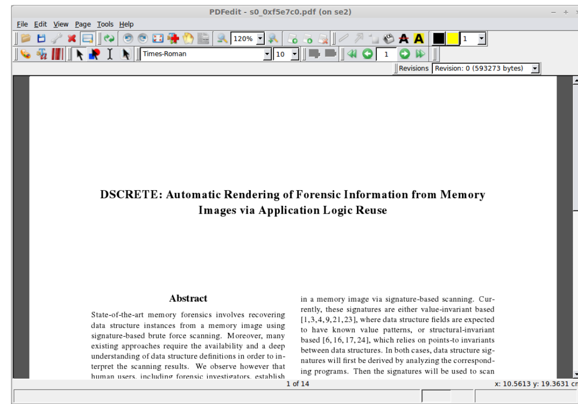
Increasingly, forensic investigators are looking to access the wealth of actionable evidence stored in a system's memory. Typically, this requires that an investigator have access to a suspected machine, prior to it being powered down, to capture an image of its volatile memory. Further, memory acquisition (both hardware [6] and software [25] based) must be as minimally invasive as possible since they operate directly on the machine under investigation. The resulting memory image is then analyzed offline using memory analysis tools. Therefore, the goal of memory analysis tools (like the work presented in this paper) is to recreate, in the forensics lab, the system's previously observable state based on the memory image.

Uncovering evidence from memory images is now an essential capability in modern computer forensics. Most state-of-the-art solutions locate data structure instances in a memory image via signature-based scanning. Currently these signatures are either value-invariant based [2, 3, 9, 21, 23, 26], where data structure fields are expected to have known value patterns, or structural-invariant based [5, 16, 17, 24], which rely on points-to invariants between data structures. In both cases, data structure signatures will first be derived by analyzing the corresponding programs. Then the signatures will be used to scan memory images and identify instances of the data structures. Contents of the identified instances will be presented to forensic investigators as potential evidence.

A significant challenge, not addressed in existing memory forensics techniques, is that investigators may not be able to *interpret the content of data structure fields*, even with the data structure's syntax and seman-



(a) Signature-based scanner output.



(b) DSCRETE-based scanner output.

Figure 1: Illustration of content reverse engineering challenge. (a) Raw content of an in-memory data structure instance representing a PDF file. (b) The same data structure after applying DSCRETE's scanner based on content reverse engineering.

tics. This is very common for data structures with application-specific encoding, such as those representing images, passwords, messages, or formatted file contents (e.g., PDF), all of which are potential evidence in a forensic investigation. For example, an investigator may know that a `buffer` field is holding a photo image (through existing data structure reverse engineering and scanning techniques [9, 15–17, 24, 26]), but still cannot display (and hence understand) the image. Similarly, a `message_body` field may hold an instant message, but the message is encoded, and hence it cannot be readily interpreted. We call this the *data structure content reverse engineering* challenge.

To enable automatic data structure content reverse engineering, we present DSCRETE[1], a system that automatically *interprets and renders* contents of in-memory data structures. DSCRETE is based on the following observation: The application, in which a data structure is defined, usually contains interpretation and rendering logic to generate human-understandable output for that data structure. Hence the key idea behind DSCRETE is to *identify and reuse* such interpretation and rendering logic in a binary program — without source code — to create a "scanner+renderer" tool. This tool can then identify instances of the data structure in a memory image and, most importantly, render them in the application's *original output format* to facilitate human perception and avoid the overhead of reverse engineering data structure signatures required by signature-based memory image scanners.

To illustrate the challenge of data structure content reverse engineering, we present a concrete example (from Section 4). Figure 1a shows the raw content of an in-memory data structure graph representing a PDF file. This is the output produced by existing signature-based scanners. For comparison, Figure 1b shows the same data structure content *after* applying DSCRETE's scanner with content reverse engineering capability. It is quite obvious that the reverse-engineered content would be far more helpful to investigators than the raw data structure content.

We have performed extensive experimentation with DSCRETE using a wide range of real-world commodity application binaries. Our results show that DSCRETE is able to recover a variety of application data — e.g., images, figures, screenshots, user accounts, and formatted files and messages — with very high accuracy. The raw contents of such data would otherwise be unfathomable to human investigators.

The remainder of this paper is organized as follows: Section 2 presents an overview of DSCRETE. Section 3 details the design of DSCRETE. Section 4 presents our evaluation results. Section 5 discusses some observations from our evaluation, current limitations, and future directions for this research. Section 6 discusses related works and Section 7 concludes the paper.

2 System Overview

2.1 Key Idea: Executable Code Reuse

DSCRETE is based on reusing the existing data structure interpretation and rendering logic in the original application binary. As a simple example, consider the Linux `gnome-paint` application. At the high-level,

[1] DSCRETE stands for "Data Structure Content Reverse Engineering via execuTable rEuse," pronounced as "discrete."

`gnome-paint` works as follows: An input image file is processed into various internal application data structures. The user then performs edits to and saves the image. To save the image, `gnome-paint` will reconstruct a formatted image from its internal data structures and write this image to the output file.

Later, if a forensic investigator wanted to recover the edited image left by `gnome-paint` in a memory snapshot, DSCRETE would be used to identify and automatically reuse `gnome-paint`'s own data structure rendering logic. First, DSCRETE will identify and isolate the corresponding data structure printing functionality within the application binary. For brevity, let us refer to this printing/rendering component as the function *P*. *P* should take as input a data structure instance and produce the human readable application output which is expected for the given data structure. In the case of `gnome-paint`, this component is the `file_save` function. It takes as input a `GdkPixbuf` structure and outputs a formatted image to a file. Note that *P* may not be a function in the programming language sense, but instead a *subsection of the application's code* responsible for converting instances of a certain data structure into some human-understandable form (e.g., output to the screen, file, socket, etc.).

Once *P* is identified, DSCRETE will reuse this function to create a *memory scanner+renderer* (or "scanner" for short) to identify all instances of the subject data structure in a memory image. If *P* is well defined for the input data structure, then one can expect *P* to behave erroneously when given input which is not a valid instance of that data structure. Under this assumption, we can present each possible location in the memory image to *P* and see if *P* renders valid output for the data structure of interest. We note that should an investigator alternatively choose to use a signature-based memory scanner to locate data structure instances, the DSCRETE-generated scanner *is still able to render* any located instances.

2.2 Overview of DSCRETE Workflow

Figure 2 presents the key phases and operations of DSCRETE. The first input is a binary application for which an investigator wishes to recover application data of interest from a memory image. To avoid compatibility issues (such as different data structure field layouts), this binary should be the same as the one that has contributed to the memory image.

The subject binary is then executed under instrumentation to identify the code region responsible for converting a specific data structure into application output (the function *P* defined earlier). We refer to this phase of DSCRETE as "tracing," and the details of this step are presented in Section 3.1. In the next phase, "identification" (Section 3.2), a *graph closure algorithm* is used to formulate a list of possible candidates for *P*. Each candidate is tested, by the "tester" component (Section 3.3), with a ground truth data structure instance to determine if it can serve as a viable memory scanner.

Once the specific application logic (*P*) is identified, DSCRETE packages this logic as a *context-free memory scanner* (Section 3.4), which will be presented to forensic investigators to scan and interpret memory images in this and future investigations involving the same application. We point out that the first three phases (tracing, identification, and tester) are a one-time procedure internal to DSCRETE and do *not* involve field investigators who will be using the DSCRETE scanners in their practice.

It is important to note that, unlike signature-based memory scanning techniques, we do not attempt to find and return the raw contents (bytes) of identified data structure instances in a memory image. Instead, we aim to present the investigator a set of *application-defined outputs* that would naturally be generated by the subject application, had it executed *P* with the data identified in the memory image. We emphasize that DSCRETE does *not* infer data structure definitions (unlike [17, 24]), nor does it derive data structure signatures (unlike [16]).

2.3 Assumptions and Setup

Firstly, we assume that when producing DSCRETE-based memory scanners (which is typically the task of a central lab of a law enforcement agency), the subject binary can be executed. This includes recreating any execution environment (i.e., operating system and application version, required libraries, directory configurations, etc.) which the application requires. We believe that this assumption is not overly difficult to realize. In a real forensic investigation, such runtime configuration information can be collected via preliminary examination of suspect or victim machines. Additionally, our dynamic instrumentation requires that address space layout randomization (ASLR) be turned off during the production of the DSCRETE memory scanners (i.e., only the investigator's personal workstation, *not* the suspect machine under study). The reason for this will become clear in Section 3.3.

Secondly, we assume that the OS kernel's paging data structures in the subject memory image are intact. This is a similar assumption made by many previous memory forensics projects [16, 21, 26]. We require this because DSCRETE takes as input only the subject application's memory session from the suspect machine. For our evaluation, we extracted the memory pages directly from running applications — which is preferred when an investigator has physical access to a suspect's ma-

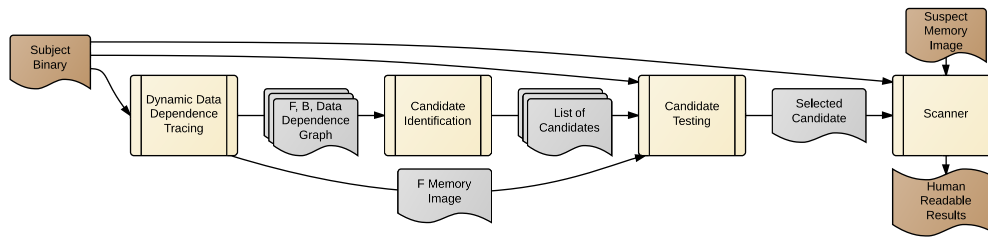


Figure 2: Overview of DSCRETE workflow.

chine. However in many forensic investigations only the memory snapshot and hard disk image are available. In this case the Volatility [26] `linux_proc_maps` and `linux_dump_map` plug-ins (or `memmap` and `memdump` for Windows) can be used to identify and extract process pages and mapping information from a whole system memory image.

3 System Design

In this section, we explain each phase of DSCRETE.

3.1 Dynamic Data Dependence Tracing

The first phase of DSCRETE, "tracing," collects a dynamic data dependence trace from the subject application binary. This trace must contain some portion of the future scanner's code (i.e., the code responsible for rendering a data structure of interest as human-understandable output). To collect this trace, we (as the central lab staff producing the scanners for field investigators) interact with the application to perform the following actions:

1) Create and populate an instance of the data structure used to store the data of interest. However, we make no assumptions on the knowledge about this data structure. We only assume that *some* data structure exists in the application which holds forensically interesting information in its fields.

2) The data structure of interest must be emitted as observable outputs (e.g., to files, network packets, or displayed on screen). This is to allow the scanner production staff to express their forensic interest by marking (part of) the output.

Again let us use `gnome-paint` to illustrate this procedure. To accomplish Step 1, we only need to execute `gnome-paint` with some input image. This will cause `gnome-paint` to create and populate numerous internal data structures to store the image's content. To accomplish Step 2, we only need to save the image to an output file. `gnome-paint` will process the image for output and call the GDK library's `gdk_pixbuf_save` function with the image's content as a parameter. While this may seem like a highly simplified example, the case studies in Section 4 show that in general we do not need to perform lengthy or in-depth interaction with an application to accomplish these two requirements.

Meanwhile, DSCRETE will be collecting each instruction's data dependence and recording any library functions or system calls invoked by the application as well as their input parameters. This is used to later identify which known external functions, specifically those which emit data to external devices, were invoked with the forensically interesting content as a parameter (`gdk_pixbuf_save` from our `gnome-paint` example). Note that since our analysis is at the binary level without symbolic information, we consider a parameter to be any memory read inside a function that depends on a value defined prior to the function's invocation. The memory may be accessed inside the function, subsequent functions, or as an argument to a system call[2], and the content read is logged as parameters. We exclude any memory not previously written to by the application or a previous library function, allowing us to ignore any memory which is private to the library function and not related to the parameter (i.e., the transformed data structure). This logging results in an output file containing the list of invoked external functions and parameters to each (similar to the Linux `strace` utility).

It is important to note here that DSCRETE saves a snapshot of the process's stack and heap memory at the invocation of any external library function which leads to an output-specific system call (i.e., `sys_write`, `sys_writev`, etc.). We (as the forensics lab staff) may, optionally, further specify individual library functions for which a snapshot should be taken. For example, if we know that the forensic evidence will be rendered on the application's GUI, then we may choose to only log visual-output related library calls in the GTK library. These snapshots will later be used to test possible *closure points* (defined in Section 3.2).

Once Steps 1 and 2 are accomplished, we may terminate the subject binary and search the log of external function calls for one in which the forensically in-

[2] We assume that system call interfaces are known and thus we can mark which parameters and memory ranges are read and which are written to.

teresting data is seen as parameters. Once suitable functions are chosen, DSCRETE only needs to identify which bytes of the parameters for those function invocations are of forensic value.

The chosen function invocations and set of parameter bytes will be important for two reasons: First, the parameter bytes will serve as the source nodes in our data dependence graph. Second, the function(s) will be used as the termination point for our scanner and the corresponding bytes will be the *output* of the scanner. For brevity, these functions will be referred to as F and the selected forensically interesting bytes of F's parameters as the set B. For our running `gnome-paint` example, consider `gdk_pixbuf_save` as F and the image buffer it prints to the output file as B.

Finally, a data dependence graph is generated using the trace gathered during dynamic instrumentation. The graph begins with the instructions responsible for computing the bytes of B as source nodes. Then in an iterative backwards fashion, any instruction which a graph node depends on is also added to the graph. Eventually, the graph will contain any instruction instance which led to the final value of B's bytes. This process is identical to that of typical dynamic slicing [13], we just chose to ignore control dependence as it is not required for identifying the functional closure (to be described next).

3.2 Identifying Functional Closure

Given F, B, and the data dependence graph, DSCRETE must find a *closure point* for the rendering function P. We define a closure point as an instruction in the data dependence graph which satisfies: 1) It directly handles a pointer to the forensically interesting data structure and 2) Any future instruction which reads a field of the data structure must be dependent on the closure point. This leads to the nice property that by changing the pointer handled by the closure point, we can change the data output by P. Returning to the `gnome-paint` example, the closure point is the instruction which moves a `GdkPixbuf` pointer into an argument register during `file_save`.

However, without source code or the effort of reverse engineering the subject binary, we cannot know the closure point for certain a priori. In fact, there may be multiple closure points in a program, any of which will satisfy our criteria above. To find a valid, usable closure point we use a combination of a graph closure algorithm and heuristics to output a list of *closure point candidates*. Each candidate is a tuple of the following: the address of an instruction which may satisfy the above criteria, the register or memory operand which it stores a pointer to, and the value of that pointer from the data dependence trace taken during tracing (Section 3.1).

Algorithm 1 Identifying *Closure Point* Candidates

```
Input: DataDepGraph(V, E), p
Output: Candidates[]
  SubGraphs[] ← ∅
  Previous_Candidate ← ∅
  for node n ∈ V in Reverse Temporal Order do
      G(Vn, En) ← ∅                          ▷ Build subgraph rooted at n
      Vn ← {n}
      for (n, t) ∈ E do          ▷ Each t that depends on n (may be ∅)
          Gt(Vt, Et) ← SubGraphs[t]          ▷ SubGraph rooted at t
          Vn ← Vn ∪ Vt
          En ← En ∪ Et ∪ (n,t)
      SubGraphs[n] ← G
      if Is_Store_Instruction(n) then            ▷ Apply heuristics to n
          val ← Stored_Value(n)
          loc ← Store_Location(n)
          if Is_Possible_Pointer(val) then
              if |SubGraph[n]|>|SubGraph[Previous_Candidate]| then
                  Candidates ← Candidates ∪ (n, loc, val)
                  Previous_Candidate ← n
      if |SubGraphs|> p%×|DataDepGraph| then
          break                    ▷ Only consider p% of DataDepGraph
```

We call this phase "candidate identification." The algorithm to identify closure point candidates is given in Algorithm 1. Starting from each byte in B, the algorithm steps through the data dependence graph in reverse temporal order (i.e., from the last instructions executed to the first). For each node visited (n) the algorithm builds a graph containing all previously visited nodes which depend on n (G in Algorithm 1). Essentially, graph G will resemble a subgraph rooted at n with its leaves accessing some bytes of B.

For each node n added to these subgraphs, the algorithm performs the following heuristic checks; any node which passes these checks is considered a closure point candidate. First, n must store a value (either to a register or memory location) which could be a possible data structure pointer (any integer value that falls within a memory segment marked readable and writable). Second, the size of the dependence subgraph rooted at n must be larger than the previous candidate's subgraph. The intuition here is that a correct closure point will take as input a pointer to a data structure instance, and store this pointer to be reused by the rendering function P. Thus for the part of the data dependence graph responsible for rendering a data structure instance, the largest subgraph must have the closure point at its root. Consider a data dependence graph for the `file_save` function from `gnome-paint`: The largest subgraph of this data dependence graph should be rooted at the input `GdkPixbuf` pointer.

Another heuristic is to stop the algorithm after only a small percent of the data dependence graph is analyzed. Note that the data dependence graph contains instructions from F back to the application's `main` function. Further, P will be close to F in the graph and signif-

icantly smaller than the rest of the application's code. This percentage is taken as a configurable input (p in Algorithm 1) and is set via a forward iterative approach. In our evaluations in Section 4, we started with a p value of 1 and incremented p until a valid closure point was found. Even in the extreme case (`top`), p was never more than 10 and was often less than 5.

In all of our evaluations, the number of candidates never exceeded 102 and was often below 30. Additionally, as will be explained in the next section, we never need to verify (or even see) any of the candidates. The testing of candidates is done mostly automatically.

3.3 Finding the Scanner's Entry Point

To test each closure point candidate, DSCRETE will run a modified version of the memory scanner described in the next section. This modified scanner, named the candidate "tester," takes as input: 1) the known end point of the scanner (i.e., F), 2) the memory image taken when F was executed, 3) the list of candidates, and 4) the subject binary. The modified scanner will treat F's memory image as the "suspect" memory image to scan. We assume that this memory image contains a valid instance of the data structure which held the data seen in B because the application was in the process of rendering/emitting this data structure instance's fields when the memory image was captured.

The candidate tester will re-execute the subject binary from the beginning, but before the process is started the scanner maps the "suspect" memory image's segments into the address space. Each segment (a set of pages) is mapped back to the address from which it was originally taken[3]. This ensures that pointers in these memory segments will still be valid in the new process's address space. Note that ASLR is disabled during DSCRETE operations. At this point, the new process is unaware of the added memory segments and executes normally using only its new allocations. Later, we will intentionally force the new process to use a small portion of the old process's data session to test closure point candidates, a technique we call *cross-state execution* (discussed in Section 3.5).

In the new execution, the forensically interesting data seen in this run of the application should be altered (e.g., executing `gnome-paint` with a different image). This will later allow the user to easily determine which candidate's output is correct (from the data structure in the memory image).

[3] We have not seen any cases where critical segments overlapped. This is because the segments are being mapped into ranges usually reserved for heap and stack space. Since these segments are almost universally relocatable the new process is simply allocated pages around our memory image.

The application runs until a closure point candidate instruction is executed. Here, the tester forks an identical copy-on-write child of the subject application to perform the actual scan; the parent process will be paused until the child has completed. The scanner looks up which register or memory operand this candidate stores its pointer value into and overwrites this location with the pointer's value stored in the candidate. Note that if this candidate is a correct closure point, then the stored pointer value is a valid pointer to a data structure instance in the mapped memory image. This assumes that the data structure instance is not corrupted from the beginning of the rendering function P (for which this candidate may be an entry point) to the invocation of F. Since all candidates are reasonably close to the invocation of F (within $p\%$ of the total trace size), we find that this is never a problem in practice.

Further, if this candidate is a correct closure point, the child process will now execute P, access the old process's memory segments (via the changed pointer value), generate the same bytes for B, and invoke F with these bytes. Imagine that, for our `gnome-paint` example, this candidate is the instruction which moves a `GdkPixbuf` pointer into a register during `file_save` (P). Now `file_save` will execute in the child process with the `GdkPixbuf` structure inside the memory image and should call `gdk_pixbuf_save` (F) with an identical image as was previously rendered (B). Also, recall that the forensically interesting information seen in the new run is altered. This is to easily partition between output generated from the memory image and output from the new execution of the application.

During testing, if the child process crashes after the pointer replacement, then the candidate is assumed incorrect and thrown out. When the F function(s) execute to completion (recall that in our `gnome-paint` example F is `gdk_pixbuf_save`) then the content given as input to F is recorded as a result for this closure point candidate test. An example of this recorded output is given later in Figure 6 (Section 4.3.1). The end of a scan is determined as follows: When F is a single function invocation, the child process is killed after F returns. If F consists of multiple invocations, the scan continues until the execution call stack returns to a point before the closure point. The parent process is then resumed, and this is repeated until all candidates are tested. A candidate is considered a valid closure point if it has accurately recreated the bytes chosen for B.

3.4 Memory Image Scanning

Once the data structure rendering function P has been identified, DSCRETE can build a memory scanning+rendering tool out of the subject binary. In fact, the production memory scanner is quite similar to the mod-

ified scanner used for testing candidates in the previous section. The difference is that we do not know where in a suspect memory image the data structures may be. The input to the memory image scanner tool are: 1) the chosen entry point and exit point of the printing function *P*, 2) the subject binary, and 3) the suspect memory image (as described in Section 2.3).

Again the scanner will re-execute the subject binary with the suspect memory segments mapped back into their original placements. Like before, the suspect memory segments will not be used until scanning begins, and until then the process executes using only its new allocations. With the same application input from candidate testing, the execution will reach *P*'s entry point, where the scanner pauses the application. For each address in the memory image, the scanner will fork an identical copy-on-write child and assign *P*'s pointer to the next address in the memory image. In essence, the scanner is executing *P* with a pointer to each byte of the suspect memory image. The scanning child process executes until *P*'s end point (as defined in the previous section) and then *P*'s output is recorded to a log or the child process crashes.

The intuition behind re-executing the application from the beginning is to automatically rebuild any dependencies required by *P*. DSCRETE requires that *P*'s only input be a pointer to a possible data structure. In reality, *P* may depend on multiple parameters set up by the application prior to the closure point. By re-executing the application from the beginning, we ensure that any other dependencies *P* has are taken care of before the scanner injects a data structure pointer.

The execution of *P* is done in a child process to isolate side effects. Not surprisingly, the vast majority of addresses will cause invalid memory accesses or other exceptions, and by scanning each byte in a separate process the scanner ensures that side effects do not contaminate future scans or global values. To speed up scanning, multiple child processes can be spawned to run in parallel.

In some rare cases, *P* is too simple (performs too little input processing) to crash on invalid input. For such cases, we allow for a user-defined post-processing phase. We still assume no use of source code or reverse engineering effort, but the user may perform sanity checks based on the known format or value ranges for an application's output. For instance, in our `top` case-study we had to remove any output which had a negative process ID or blank user or process name field. In our experiments, only three cases — `CenterIM`, `top`, and `Firefox VdbeOp` — required any post-processing. Further, this only occurs for very simple textual *P* functions — complex cases such as those requiring content reverse engineering naturally involve more strict parsing and input sanitization.

3.5 Cross-State Execution

DSCRETE maps one process's address space into the address space of another. Further, when DSCRETE executes the function *P*, this code will evaluate data in both the old and new address spaces. Once DSCRETE replaces a data structure pointer at the closure point, the scanning process will then access fields from the data structure in the old address space while still using stack and other heap objects in the new address space.

Ideally, any sub-execution that depends on the closure point would exclusively access the state from the old address space. In other words, we expect the continuation after the pointer replacement would consist of two disjoint sub-executions, one corresponding to running *P* on the old address space and the other corresponding to the rest of the execution exclusively on the new address space. However, due to the complex semantics of real world programs, such separation may not be achievable. There are two possible problems: 1) An instruction execution may depend on state from both address spaces, resulting in some state that is infeasible in either the original or the new execution. We call such instructions *confounded instructions*. 2) Since the old memory snapshot may not be complete, an instruction may access a location in the old space that is not mapped in the new space. Note that this location may now correspond to a valid address in the new space such that the access becomes one to the new space. We call this a *trespassing instruction*. Both could cause crashes and hence false negatives.

Consider the example in Figure 3. Figure 3a shows two functions[4]. The first function (lines 2 - 6) creates a `Color` object and adds it to the `color_cache`. The other (lines 7 - 14) renders a window, including drawing the `Color` to a frame and emitting the window title as a string. Note that different executions may add different `Color` objects to the cache. Specifically, the number of `Color` objects and their order vary across executions. Later, the window rendering function will look up a `Color` object from the cache using its id.

Assume we (as forensics lab staff) mark the `EmitString()` function at line 13 as *F* and `s` (the window title) as *B*. Following the candidate identification algorithm, we compute the backward data-dependence of *B* as those boxed statements. We further identify line 8 as the closure point candidate.

However, when we test this candidate, cross-state execution leads to undesirable results if not properly handled. Let us assume that two `Color` objects were cached during the original execution, whereas only one `Color` is added in the candidate test execution. Figure 3b shows the trace of the candidate test execution on the left, and,

[4] Our discussion is at the source code level for readability, whereas our design and implementation assume only the application binary.

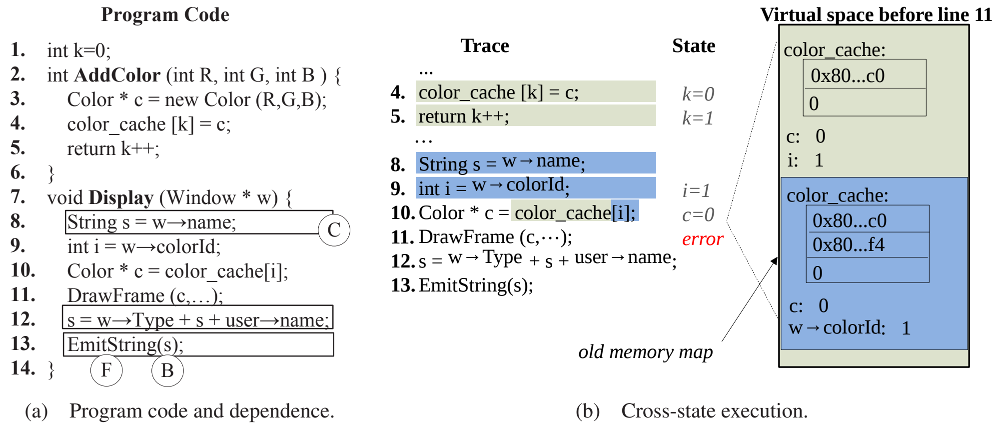


Figure 3: Example for cross-state execution.

on the right, it shows the state of the new address space right before the execution of line 11. Note that the pages of the old address space are mapped inside the new address space. Each executed statement in the trace is colored based on the address space it operates on. Particularly, lines 4 and 5 execute before the pointer `w` is replaced at line 8, and hence belong to the new space. In contrast, lines 8 and 9 belong to the old space, as their values are loaded from locations derived from the replaced `w`. Line 10 is a confounded instruction, as the array `color_cache` belongs to the new space while `i` belongs to the old space. As a result, an invalid color is loaded, leading to a crash. However, observe that lines 10 and 11 are not in the data dependence of B, as such we could potentially skip them.

Therefore, given a closure point candidate C and its termination point F, DSCRETE scans the original execution trace from C to F during the candidate identification phase. For each address dereference it encounters, it tests if the address is exclusively dependent on the pointer parameter at C. If not, it is a confounded dereference. DSCRETE further tests if the dereference is in the data-dependence graph of B, and if not, marks the instruction as an irrelevant dereference to be skipped during test execution and later scanning executions. In practice, we observed confounded memory dereferences in only one of the cases we studied.

Handling trespassing instructions is relatively easier. Given a closure point candidate C and its termination point F, DSCRETE scans the original execution trace from C to F and marks each address dereference that it encounters and is dependent on the pointer parameter at C. At runtime, if a marked dereference accesses a location in the new space, it is a trespassing access and can be skipped.

4 Evaluation

DSCRETE leverages the PIN binary analysis platform [19] to perform instrumentation. Since PIN executes before the subject binary is loaded, this allows us to map the memory image into the new process's address space before the operating system's loader can claim stack and heap regions. DSCRETE relies on minimal OS-specific knowledge (i.e., system call and application binary interface definitions), thus DSCRETE can easily be ported to any operating system that PIN supports. In the remainder of this section, we present results from evaluating DSCRETE with a number of real-world applications and focus on a subset which highlight the use of DSCRETE and a few critical observations.

4.1 Experimental Setup

Our evaluation used a Ubuntu 12.10 Desktop system as the "suspect" machine. Each application was installed on the machine and interacted with by the authors to generate sufficient allocations and deallocations of data structures. We used `gdb` to capture memory images from the application periodically during the system's use. To attain ground truth, we manually instrumented the applications to log allocations and deallocations for data structures corresponding to the output of forensic interest (i.e., B in Section 3.1). This log was later processed to measure false positives (FP) and false negatives (FN). For analysis, we employed a Ubuntu 12.10 virtual machine. To recreate the suspect machine's running environment, we copied the applications and needed configuration files from the suspect machine's hard disk. We then performed all forensic investigation within the virtual machine.

Application	F	Forensically Interesting Data	Size B (bytes)	p%	#C	#O	#P
CenterIM	`SSL_write`	Username & Password	336	5%	46	1	1
convert	`fwrite`	Output Image Content	81902	9%	18	7	2
gnome-paint	`gdk_pixbuf_save`	Image Content	670900	1%	18	2	2
gnome-screenshot	`gdk_pixbuf_save_to_stream`	Screenshot Content	1139791	1%	5	4	3
gThumb	`gtk_window_set_title`	File Info Window Title	85	1%	102	4	2
	`gdk_pixbuf_save_to_bufferv`	Image File Content	20360	1%	10	3	3
Nginx	`write`	HTTP Access Log	181	5%	25	1	1
PDFedit	`fwrite, fputc`	Edited PDF Content	30416	1%	46	6	3
SQLite3 Shell	`fputs`	Database Query Results	19	2%	4	1	1
	`fprintf`	Database Op. Log	38	2%	17	5	1
top	`putp`	Process Data	132	10%	1	1	1
Xfig	`fprintf`	Figure Content	1001	1%	9	3	3

Table 1: Results from identifying applications' P functions (#C shows the number of identified candidates, #O shows how many of those produced output, and #P shows the final subset which are valid closure points).

4.2 Function Identification Effectiveness

This section presents results of isolating the data rendering function P in each tested application. From the `CenterIM` instant messenger, we target the component which emits the user's login and password (still in plain text) to an SSL socket. Also, given the importance of image content to investigations, we isolate image rendering functions from three common image editors: `convert`, `gnome-paint`, `gThumb`, as well as the `gThumb` GUI function which displays the current image's name to the window title. The output function of `gnome-screenshot` can allow an investigator to see what screen-shot a suspect was capturing. Additionally, we reuse `Xfig`'s figure saving P function to reconstruct a vector figure that was being worked on. As we introduced in Section 1, the PDF saving functionality of `PDFedit` allows investigators to recover the edited PDF file. For internal application data, we identified P functions for `SQLite`'s query results and operations log (more on how these scanner+render tools are used later in this section). It is very common for attackers to tamper with server log files, so we isolated the `Nginx` webserver's connection logging function, thus an investigator can compare with the uncovered in-memory connections. Finally, for details on all running processes in a suspect system, we identified the process data printing logic in the `top` utility.

Table 1 shows a summary of the results from each of these applications. The application name and F function are shown in Columns 1 and 2 respectively. Column 3 details the forensically interesting data that were to be emitted by $F(B)$ and Column 4 shows the size of B in bytes. The percentage of the data dependence graphs used to generate candidates is shown in Column 5. Finally Columns 6 to 8 show the number of candidates identified by our algorithm (#C), how many of those produced any output (#O), and the final subset which accurately recreated B and could be used for valid closure points (#P), respectively.

From Table 1 we make the following observations: First, our algorithm/heuristics used to identify closure point candidates are accurate enough to limit the number of candidates to a reasonable search space. Although candidates are tested automatically during the candidate tester's execution, we aim to minimize the number of candidates to test. From Table 1, we see that 11 out of the 12 applications have less than 50 candidates. The only application with more than 50, `gThumb`, has 102, and as we see in Row 5 of the table, they are drastically narrowed down by the candidate tester. Manual investigation revealed that `gThumb`'s larger number of candidates was due to extra data dependencies caused by another parameter to its F function (`gtk_window_set_title`).

The second observation we make is that, of the total number of candidates identified, very few will be true closure points. This is intuitive since there is only one true entry to the P function in the application. Third, since the number of candidates which produced valid output is so small, it is relatively simple for a DSCRETE user to identify which candidate accurately reproduced B.

On average, each candidate testing component rendered application output for only three closure point candidates. The maximum, `convert`, rendered only seven outputs during candidate testing. Further, *more than half* of the applications produced ideal candidates — all candidates that rendered output were valid candidates. For the other five applications, about 45% of candidates

which produced output accurately recreated the expected forensically interesting data (i.e., the new output matched that seen before). This shows that: 1) Visually inspecting candidate output is a reasonably quick and practical task and 2) DSCRETE can identify and validate closure point candidates with high accuracy.

Table 1 shows that it is not uncommon for multiple correct closure points to exist for a *P* function. Manual investigation revealed that this is caused by two program features: nested data structure pointers and register-to-stack spilling. In the nested data structure situation, if a data structure A has a pointer to structure B and *P* uses the B pointer within A, then either the A pointer or its internal B pointer may be valid closure points for *P*. For the register-to-stack spilling situation, a pointer to an input data structure is initially stored in a register, but when contention forces that register to be spilled onto the stack, either the initial register or its later stack-saved location may be used for closure points.

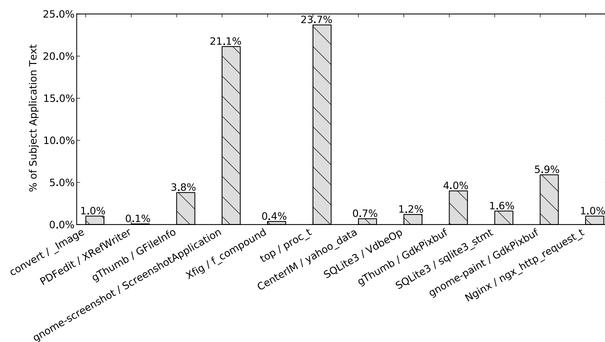


Figure 4: Normalized size of *P* vs. entire binary code.

Table 1 also shows that a valid closure point is typically located in the bottom 5% of the data dependence graph. Thus, the actual rendering function being reused is often only a small percentage of the binary's text. Figure 4 shows the normalized percentage of the host binary which we reuse for each scanning function. The size of the reused code is measured as the total in-memory size of all unique instructions observed during all re-executions of *P*. `Top`, `gnome-screenshot`, and `gnome-paint` are outliers due to the relatively small size of the applications and the resulting dependence graphs.

SQLite *P* Functions. An interesting application of DSCRETE can be seen in the experiments with SQLite. For these experiments DSCRETE was used with the SQLite3 command shell and a homemade database file to find *P* functions for a database query's result (`struct sqlite3_stmt`) and operations log (`struct VdbeOp`). These data structures are defined by the SQLite3 library and exported to client applications. The *P* functions DSCRETE identifies would be used to build memory scanner+renderer tools which could discover those data structures and render their content in the same format as the SQLite3 command shell.

These scanners could then be used on memory images from *any application which uses SQLite*. Since these data structures are defined by the SQLite library, any application using SQLite should transitively allocate and use these data structures. Further, we are reusing the SQLite3 command shell's *P* functions, so even if an application never outputs the data held in these structures, we can still discover and interpret them using the more general SQLite memory scanners. In the next section, we show results from applying these scanner+renderer tools to memory images from Mozilla Firefox and darktable image editor.

4.3 Memory Scanner Effectiveness

Table 2 reports the effectiveness of the DSCRETE-generated scanner+renderer tools when scanning a context-free memory image from each application. The application name is shown in Column 1. The subject data structure (input to *P*) and the structure's size are shown in Columns 2 and 3[5]. The number of true instances in the suspect memory image is shown in Column 4[6]. Column 5 shows the total number of output generated by each scanner+render tool. Columns 6 to 10 show the number of generated output which are: true positives (TP) - backed by true data structure instances, false positives (FP) and the percentage of FPs in the total output (FP%), and false negatives (FN) and the corresponding FN percentage.

This table shows that the *P* function identified by DSCRETE is almost always well defined. This allows DSCRETE to uncover and render valid data structure instances with 100% accuracy for most cases. Specifically, Table 2 shows that DSCRETE's scanner+renderer tools are perfectly accurate (i.e., no FP and no FN) in 11 out of the 13 cases. We analyze the two FP/FN cases in detail later in this section. More importantly, DSCRETE overcomes the data structure content reverse engineering challenge by displaying the results in each application's original output format. The test cases covered in Table 2 span a wide range of application data: usernames and passwords, images, PDF files, vector-based graphics, as well as formatted and unformatted textual output. This portrays the generality of DSCRETE and represents several key types of evidence that would be very difficult

[5]Such information was obtained via manual instrumentation, inspection, and reverse engineering only for the purpose of evaluation. DSCRETE does not need or have access to this information during operation.

[6]This includes all the data structure instances which were allocated and not yet released and overwritten when the memory image was captured.

Application	Subject Data Structure	Size (bytes)	True Instances	Total Output	TP	FP	FP%	FN	FN%
CenterIM	`yahoo_data`	160	1	1	1	0	0.0%	0	0.0%
convert	`_Image`	13208	1	1	1	0	0.0%	0	0.0%
darktable	`sqlite3_stmt`	272	1	1	1	0	0.0%	0	0.0%
Firefox	`sqlite3_stmt`	272	1	1	1	0	0.0%	0	0.0%
	`VdbeOp`	24	788	1384	753	502	40%	35	4%
gnome-paint	`GdkPixbuf`	80	51	51	51	0	0.0%	0	0.0%
gnome-screenshot	`ScreenshotApplication`	88	1	1	1	0	0.0%	0	0.0%
gThumb	`GFileInfo`	48	382	381	381	0	0.0%	1	0.4%
	`GdkPixbuf`	80	63	63	63	0	0.0%	0	0.0%
Nginx	`ngx_http_request_t`	1312	6	6	6	0	0.0%	0	0.0%
PDFedit	`XRefWriter`	344	1	1	1	0	0.0%	0	0.0%
top	`proc_t`	720	382	382	382	0	0.0%	0	0.0%
Xfig	`f_compound`	112	1	1	1	0	0.0%	0	0.0%

Table 2: Results from DSCRETE-generated scanner+renderer tools.

(if at all possible) to reconstruct from raw data structure contents.

Table 2 shows that many of the subject data structures are smaller than the resulting application output (B from Table 1). Our manual analysis of these structures reveals that 10 of the 12 data structures contain several pointers to other data structures used by P. This confirms our intuition that, in order to recover usable evidence from a memory image, numerous data structures must be uncovered and interpreted. Note that an investigator never actually sees any of these structures, but rather is presented only the application output rendered from the structures' contents. Figure 1a in Section 1 is one such example.

Another metric to report is the time taken to scan, which varies depending on: 1) the complexity of the rendering function P and 2) the size of the memory image being scanned. Figure 5 shows the scanning speed in bytes-per-second for each scanner function in our evaluation. During our experiments, the size of the applications' heaps ranged from 400KB to about 5MB, and total heap scanning time ranged between 15 minutes to just over 2 hours, with most taking about 30 to 45 minutes. Admittedly the scanning and rendering of evidence is slower than typical signature-based memory scanners, but still well within the typical time taken to process digital evidence, with the added benefit that evidence is presented in a human-understandable form. Ayers [1] points out that it may take "several hours or even days when processing average volumes of evidential data," which is confirmed by our collaborators in digital forensics practice.

False Positive and False Negative Analysis. We notice that only the gThumb and Firefox `VdbeOp` experiments experienced any negative results. Manual investigation into these two experiments' false negative results (i.e., true data structure instances not discovered by DSCRETE) revealed that those structures were allocated, but did not contain enough data to be rendered by P. They were either in the process of initialization or deletion or being used as empty templates by the application.

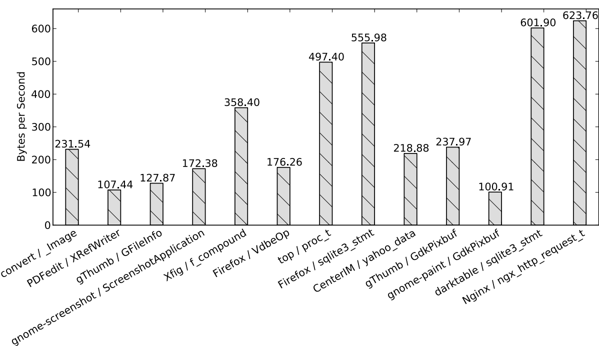


Figure 5: Observed throughput of each scanner.

Interestingly, the Firefox `VdbeOp` case study (SQLite's operations log structure) represents a counter-example to our hope that P be well defined. In this case, P performs little parsing and no sanity checks on its input. A `VdbeOp` structure is essentially a set of seven integer values, and SQLite3 uses these integers as indices in a global string table, without any sanity checks. Since this P function performs such trivial parsing, a large number of false inputs produce typical SQLite3 Shell output. We consider this a worst-case scenario for DSCRETE, and believe it is also the case for many other memory forensic techniques when facing such a trivial data structure.

In Section 1 we introduced one example of forensic

data which would be uninterpretable without data structure content reverse engineering. The complex multi-level data structure representing a PDF file requires non-trivial processing to locate the fields which contain any usable PDF content. Further, many fields are encoded, compressed, or computed only when outputting the PDF file. In the remainder of this section, we present several other application case studies with DSCRETE.

4.3.1 Case Study: `convert`

This case study highlights DSCRETE's content reverse engineering capability for image data structures. The `convert` utility is used to apply various transformations to an image file. The source image file is processed and converted into internal data structures, (i.e., an `_Image` and array of `_PixelPacket` structures). Various transformations (such as scaling, blurring, etc.) are applied, and the pixels are re-composed into an image and written to a file. It would be considerably difficult to reconstruct the image from its in-memory representation, even with a deep understanding of these structures' syntax and semantics. However, DSCRETE is able to overcome this challenge by identifying and reusing the image output component (function `WriteImage`) which constructs an output image file from an input `_Image` structure.

As shown in Row 2 of Table 1, *B* (the image's content) was seen as an argument to the `fwrite` function. Using this, DSCRETE identified 18 closure point candidates in the bottom 9% of the data dependence graph. Of these candidates, 16 clustered around the handling of `_PixelPacket` structures in the image reconstruction routine, and the remaining 2 candidates handled the input `_Image` structure at the entry to the `WriteImage` function.

The DSCRETE candidate tester component eliminated 16 candidates which handled `_PixelPacket` structures. For the remaining two candidates, DSCRETE produced the log and application output shown in Figure 6. From Figure 6 we see that Candidates 1 and 2 successfully executed *P* (ending with `fwrite`). More importantly, DSCRETE accurately rendered the `_Image` data structure's content – presenting proof that both candidates form valid *P* functions which can reconstruct the image seen previously. As Table 2 shows, this *P* function was well-defined and the resulting scanner located and rendered the "image of interest" in the memory image with no false positives or false negatives.

4.3.2 Case Study: `Xfig`

The second case study is with `Xfig`, in which data content reverse engineering is essential to uncovering usable evidence from data structure instances. `Xfig` is a Linux-based vector graphics editor which defines several types of data structures for different drawable shapes (i.e., ellipse, spline, etc.). From `Xfig`, we intended to build a scanner+renderer tool to reveal the figure a suspect was drawing. Referring back to Table 1, DSCRETE located 9 closure point candidates in the bottom 1% of the data dependence graph. DSCRETE tested these 9 candidates and decided that 3 of them which rendered output were valid closure points. One of those was chosen (DSCRETE prefers the closure point highest in the dependence graph) to build a scanner+renderer for `Xfig`'s `f_compound` data structure.

```
Candidate 1 ===== Scanning from 0x6a16c0:
  fwrite@libc ( 0x6ba360 ["<89>PNG<0d0a>"...], 1, 81902, 0x6b7320 [data] )
    Arg 1 written to file "c1_0x6ba360.out"

Candidate 2 ===== Scanning from 0x6a5c90:
  fwrite@libc ( 0x6ba360 ["<89>PNG<0d0a>"...], 1, 81902, 0x6b7320 [data] )
    Arg 1 written to file "c2_0x6ba360.out"
```

(a) Candidate test result log.

(b) Output image file for Candidate 1.

Figure 6: Candidate testing output. (a) Each *P* function is shown, similar to the Linux strace utility, with parameters seen during invocation. If the tester component is set for file output, the file name is also printed. (b) Shows the output file for Candidate 1.

An `f_compound` structure is a container for several shape structures. Each shape structure stores its dimensions, coordinates, color, etc. In order to reconstruct a figure, each of these shape structures must be recovered from a memory image, interpreted, and shape-specific rendering functions must be invoked. Existing signature-based memory scanners could present an investigator with a list of shape data structures instances from a memory image, but without the interpretation logic and shape-specific rendering, the investigator cannot see what the figure looks like. By comparison, the DSCRETE-generated scanner+renderer can locate the figure's `f_compound` structure, traverse all the contained shape structures (in the *P* function), and output `Xfig`'s original figure content. Table 2 shows that this *P* function is well-defined and recovered the figure's content with 100% accuracy from the target memory image.

4.3.3 What You Get Is More Than What You See

We observe that some applications will construct more data structures than they intend to display. Without content reverse engineering, these extra data structures would all need to be manually interpreted for investigation. DSCRETE intuitively renders such additional evidence, allowing an investigator to quickly determine if it is forensically valuable.

In our experiment with `top`, the true number of `proc_t` instances is 382, whereas while executing `top` only 31 processes were displayed at a time. Since all 382 `proc_t` structures were in `top`'s memory image, DSCRETE was able to uncover and present each as they would have been displayed by the original `top` process.

Another example is `gThumb`, which displays an image being edited and other images in the same directory. `gThumb`'s memory contained valid data structures for 63 images: 56 GUI icons and 7 suspect images, and DSCRETE recovered them all, including the 7 suspect images. More importantly, 3 of the 7 suspect images were not being displayed by the GUI. Without DSCRETE, determining which raw data structures were icons and which were evidence would require extensive manual effort. With DSCRETE, an investigator can immediately see the distinction. Note also, that those GUI icons are not false positives. Instead, they are valid and relevant image data structures, because the investigator may use such GUI artifacts to infer which application screen the suspect was focusing on.

5 Discussion and Future Work

Still at its early stage, DSCRETE represents a new approach for digital evidence collection. The prototype presented here has several limitations that will be addressed in our future work.

As mentioned in Section 3.5, cross-state execution may cause conflicting memory access patterns (i.e., confounded or trespassing instructions). DSCRETE selectively skips unnecessary instructions which may cause cross-state conflicts. However, this method is limited to the instructions recorded during tracing, and cannot reason about instructions that *were not executed*. Although we did not encounter such complications in our experiments, we do believe that they exist and will explore using static dependence analysis in the future.

DSCRETE relies on each application's own rendering logic to differentiate between valid and invalid input (data structures to be rendered). As we see in Section 3.4, this can be problematic if the rendering function performs very little input processing and validation. Our experiment suggests that this problem exists for highly simplified data structures, which may still be of forensic value. Handling such data structures is our ongoing work. Additionally, since DSCRETE reuses application binary logic, an interesting problem is to handle data which contains *exploits* against the rendering logic.

Another current limitation which we leave for future work is replacing multiple input data dependencies for a rendering function. Currently, DSCRETE identifies and replaces only a single data structure pointer seen as input to P. However, it is assumable that a single application output be generated from *multiple* unrelated data structures. Although we have not encountered such need, the problem is realistic and requires enhancements to the closure point identification and the scanning algorithms.

Like many binary analysis-based tools, DSCRETE is not yet ready to handle self-modifying code or binaries with highly obfuscated control flows, which may cause problems in dependence detection or state crossing. However, these problems are common in malware programs and hence worth solving. One future direction is to develop DSCRETE on an obfuscation-resistant binary analysis platform (e.g., [29]).

The methodology used in DSCRETE is designed to operate directly on a target machine binary. As such, it is not applicable to programs written in *interpreted* languages (e.g., Java). Such programming languages add layers of indirection between the machine instructions observed by DSCRETE and the application's true syntax and semantics (i.e., data structures and rendering functions). Developing new techniques to handle programs written in interpreted languages is an intriguing direction for our future research.

6 Related Work

Memory Image Forensics. Previous research in memory forensics has mainly centered around uncovering data structure instances using signature-based brute force scanning. Such techniques can be roughly classified into value-invariant based [2,3,9,21,23,26,27] and structural-invariant based [5,15,16].

Value-invariant signatures seek to classify data structures by the expected value(s) of their fields. More recently, DEC0DE [27] enhances value-based signatures with probabilistic finite state machines to recover evidence from smartphones. Structural-invariant based signatures are derived by mapping interconnected data structures. SigGraph [16] uses such signatures for brute-force memory image scanning. Later, DIMSUM [15] attempts to probabilistically locate data structure instances in un-mappable memory. Further, numerous forensic tools and reverse engineering systems [7, 14, 17, 20, 29] make use of data structure traversal.

Compared with these techniques, DSCRETE does not require data structure definitions or data structure field

value profiles as input. Moreover, DSCRETE can intuitively interpret data structure contents (e.g., rendering an image in memory). To the best of our knowledge, no existing memory forensic tool has similar capability.

Binary reverse engineering techniques [14, 17, 24] can reverse engineer data structure definitions (e.g., field types) from binaries. They can also reverse engineer semantic information to a certain extent. As such, they can be used in forensic analysis. However, these techniques can only reverse engineer semantics of generic data such as timestamps and IP addresses. Such approaches are hardly applicable to interpreting contents of application-specific and encoded data structure fields.

Binary Component Identification and Reuse. At the heart of the DSCRETE technique is application logic reuse. DSCRETE uses dynamic binary program tracing to identify which functional component of a binary application is responsible for generating forensically interesting output. It hence shares some common underlying techniques with existing binary identification and reuse techniques [4, 12, 18] and program feature identification [11, 28].

Similar to how DSCRETE employs a data dependence graph, Wong et. al. [28] use program slicing to identify the code region for a program feature. To further understand which application components contribute to an observed runtime behavior, Greevy et al. [11] use feature-driven dynamic analysis to isolate computational units of an application. In contrast, DSCRETE uses only an application's data dependence to identify candidates for later construction of a memory scanner+renderer.

Binary Code Reutilization (BCR) [4] involves using a combination of dynamic and static binary analysis to identify and extract malware encryption and decryption functions. The goal of BCR was to reuse such extracted logic as a functional component in a different program developed by the user. Inspector Gadget [12] uses dynamic slicing to identify specific malware behavior for extraction and later reuse/analysis. Lin et al. [18] suggested using dynamic slicing to identify applications' functional components to compose reuse-based trojan attacks. DSCRETE does not aim to extract application logic from a target binary, but rather re-execute it in-place to scan a memory image and render subject data structure contents.

Virtuoso [8] involves using dynamic slicing to identify logic from in-guest applications which could be reused for virtual machine introspection. However, Virtuoso is not able to handle input that is not encountered during off-line training. A DSCRETE-generated scanner can handle any input that the original *P* function could handle. Later, VMST [10] and Hybrid-Bridge [22] use system-wide instruction monitoring to allow introspection of one VM's kernel data from another. VMST redirects memory accesses for every instruction of the reused logic, whereas DSCRETE only needs to replace the data structure pointer at the closure point. Further, VMST relies on system call definitions to start logic reuse, while DSCRETE must automatically identify such a starting point (i.e., the closure point) in the subject binary.

7 Conclusion

We have presented DSCRETE, a data structure content reverse engineering technique which reuses application logic from a subject binary program to uncover and render forensically interesting data in a memory image. DSCRETE is able to recreate intuitive, human-observable application output from the memory image, without the burden of reverse engineering data structure definitions. Our experiments with DSCRETE show that this technique is able to effectively identify interpretation/rendering functions in a variety of real-world applications, and DSCRETE-generated scanner+renderer tools can uncover and render various types of data structure contents (e.g., images, figures, and formatted files and messages) from memory images with high accuracy.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. We are grateful for the suggestions and guidance from Dr. Golden G. Richard III. This research was supported, in part, by NSF under Award 1049303 and DARPA under Contract 12011593. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] AYERS, D. A second generation computer forensic analysis system. *Digital Investigation 6* (2009), 34–42.

[2] BETZ, C. Memparser forensics tool. `http://www.dfrws.org/2005/challenge/memparser.shtml`, 2005.

[3] BUGCHECK, C. Grepexec: Grepping executive objects from pool memory. In *Proc. 6th Annual Digital Forensic Research Workshop (DFRWS)* (2006).

[4] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *Proc. 17th Annual Network and Distributed System Security Symposium* (2010).

[5] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proc. 16th ACM Conference on Computer and Communications Security* (2009).

[6] CARRIER, B. D., AND GRAND, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1* (2004), 50–60.

[7] CASE, A., CRISTINA, A., MARZIALE, L., RICHARD, G. G., AND ROUSSEV, V. Face: Automated digital evidence discovery and correlation. *Digital Investigation 5* (2008), 65–75.

[8] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proc. 2011 IEEE Symposium on Security and Privacy* (2011).

[9] DOLAN-GAVITT, B., SRIVASTAVA, A., TRAYNOR, P., AND GIFFIN, J. Robust signatures for kernel data structures. In *Proc. 16th ACM Conference on Computer and Communications Security* (2009).

[10] FU, Y., AND LIN, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proc. 2012 IEEE Symposium on Security and Privacy* (2012).

[11] GREEVY, O., AND DUCASSE, S. Correlating features and code using a compact two-sided trace analysis approach. In *Proc. 9th European Conference on Software Maintenance and Reengineering* (2005).

[12] KOLBITSCH, C., HOLZ, T., KRUEGEL, C., AND KIRDA, E. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proc. 2010 IEEE Symposium on Security and Privacy* (2010).

[13] KOREL, B., AND LASKI, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (1988), 155–163.

[14] LEE, J., AVGERINOS, T., AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs. In *Proc. 18th Annual Network and Distributed System Security Symposium* (2011).

[15] LIN, Z., RHEE, J., WU, C., ZHANG, X., AND XU, D. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. 19th Annual Network and Distributed System Security Symposium* (2012).

[16] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND JIANG, X. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. 18th Annual Network and Distributed System Security Symposium* (2011).

[17] LIN, Z., ZHANG, X., AND XU, D. Automatic reverse engineering of data structures from binary execution. In *Proc. 17th Annual Network and Distributed System Security Symposium* (2010).

[18] LIN, Z., ZHANG, X., AND XU, D. Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *Proc. 2010 IEEE/IFIP International Conference on Dependable Systems and Networks* (2010).

[19] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices* (2005), vol. 40.

[20] MOVALL, P., NELSON, W., AND WETZSTEIN, S. Linux physical memory analysis. In *Proc. USENIX Annual Technical Conference, FREENIX Track* (2005).

[21] PETRONI JR, N. L., WALTERS, A., FRASER, T., AND ARBAUGH, W. A. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3* (2006), 197–210.

[22] SABERI, ALIREZA FU, Y., AND LIN, Z. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proc. 20th Annual Network and Distributed System Security Symposium* (2013).

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

[24] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proc. 18th Annual Network and Distributed System Security Symposium* (2011).

[25] SYLVE, J., CASE, A., MARZIALE, L., AND RICHARD, G. G. Acquisition and analysis of volatile memory from android devices. *Digital Investigation 8* (2012), 175–184.

[26] THE VOLATILITY FRAMEWORK. Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`.

[27] WALLS, R., LEVINE, B. N., AND LEARNED-MILLER, E. G. Forensic triage for mobile phones with dec0de. In *Proc. USENIX Security Symposium* (2011).

[28] WONG, W. E., GOKHALE, S. S., AND HORGAN, J. R. Quantifying the closeness between program components and features. *Journal of Systems and Software 54*, 2 (2000), 87–98.

[29] ZENG, J., FU, Y., MILLER, K. A., LIN, Z., ZHANG, X., AND XU, D. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proc. 20th ACM Conference on Computer and Communications Security* (2013).