Characterizing Data Structures for Volatile Forensics

Ellick Chan, Shivaram Venkataraman
and Nadia Tkach, Kevin Larson
Department of Computer Science
University of Illinois, Urbana-Champaign
{emchan,venkata4,ntkach2,klarson5}@illinois.edu

Alejandro Gutierrez
Graduate School of Library and Information Science
University of Illinois, Urbana-Champaign
agutier3@illinois.edu

Roy H. Campbell
Department of Computer Science
University of Illinois, Urbana-Champaign
rhc@illinois.edu

***Abstract*—Volatile memory forensic tools can extract valuable evidence from latent data structures present in memory dumps. However, current techniques are generally limited by a lack of understanding of the underlying data without the use of expert knowledge. In this paper, we characterize the nature of such evidence by using deep analysis techniques to better understand the life-cycle and recoverability of latent program data in memory.**

We have developed Cafegrind, a tool that can systematically build an object map and track the use of data structures as a program is running. Statistics collected by our tool can show which data structures are the most numerous, which structures are the most frequently accessed and provide summary statistics to guide forensic analysts in the evidence gathering process. As programs grow increasingly complex and numerous, the ability to pinpoint specific evidence in memory dumps will be increasingly helpful. Cafegrind has been tested on a number of real-world applications and we have shown that it can successfully map up to 96% of heap accesses.

I. Introduction

Traditional forensic tools gather evidence from persistent storage devices such as hard drives. In contrast, newer forensic tools also collect ephemeral evidence from the memory of a running computer. Tools such as Volatility [24] deconstruct and decipher the raw memory dumps and search for evidence of interest. While these tools can find evidence such as the process list, open network sockets and open files, which are directly related to the running system, they are often unable to provide deep semantic insight into the internal operations of the running programs. Without the use of time-consuming manual analysis or specifically developed tools, the forensic investigator cannot temporarily access or decipher all of the relevant evidence.

Current techniques to extract evidence from these memory dumps include the use of debuggers, fast file carving techniques [15] and raw pattern recognition tools such as `grep`, `strings`. Entropy analysis tools are also used to identify encrypted data. Some tools such as KOP [6] can provide an object map of kernel objects, but often times they require expert knowledge and only work on static memory dumps.

In this paper, we attempt to characterize the nature of forensic data empirically by using a type tracking system. We have developed a tool, Cafegrind[1], which monitors a running program to track the usage of data structures. By tracing the operations of dynamic memory allocations, Cafegrind infers the types of data structures created and accessed in a program. Furthermore, Cafegrind generates information that can be analyzed off-line to infer additional properties about the life-cycle of data in a program.

Our measurements include:

1) How long data structures last in memory before they are freed or clobbered
2) Which data structure types are the most frequently accessed
3) Which functions allocate and access which data types the most
4) Modification/write velocity of a data type

We have measured the data life-cycle of many real-world applications including web browsers and word processors. Our experiments show that Cafegrind can accurately map up to 96% of heap accesses and we present some case studies detailing our experiences. First, we present an analysis of how our measurements help characterize important quantities such as forensic blurriness [22], [9]. Furthermore, by observing the internal operations of programs such as web browsers, we test the effectiveness of privacy measures such as ”private browsing mode” against core dump analysis and volatile memory forensics.

Our contributions include:

1) A method and apparatus to track objects present in the memory of a running program
2) A study of the emergent characteristics of data structures including the lifetime and access patterns
3) Metrics to help forensic analysts understand the fidelity of various data types collected from a memory dump

The rest of this paper is organized as follows: We discuss existing work in volatile memory forensics in Section II. The design of Cafegrind, the type inferencing technique used and implementation details are presented in Section III. We present details of our experiments in Section IV and discuss some of the implications of our work for volatile memory forensics in Section V.

II. Related Work

Drepper [14] provides a fairly comprehensive description of the entire memory hierarchy, which extends from the design of the high-level memory allocator to the design of the underlying physical semiconductors. We highly recommend reading this work to better understand the technical issues

[1] Cafegrind: **C**/C++ **A**nalysis **F**orensic **E**ngine for (Val)grind

regarding volatile memory forensics. Gutmann [16] explored memory remanence in semiconductors early on. Chow [11], [12] performed similar measurements of data lifetime in the context of whole system simulation. Our work differs in that we perform more precise type tracking as opposed to byte-level taint tracking.

From a forensics perspective, a number of tools can collect memory dumps including the built-in core dump facility present in many operating systems. In order to collect a full system dump, crash dumps can be triggered through special debug facilities [20]. Live dumping tools include the use of the standard Unix *dd* tool and a number of opensource/commercial equivalents. Once a memory dump or core dump is obtained, analysis tools can be used to extract evidence from them. If the target application is compiled with debug information, core dumps are fairly straightforward to analyze with standard debugging and development tools. If not, then more advanced techniques such as file carving [15] may help extract specific types of data out of the memory image. Full system memory dumps can be analyzed using tools such as Volatility [24], or mapped by using tools such as KOP [6]. Amari [3] provides a good overview of existing volatile memory forensic techniques.

The subject of data persistence has been studied in the context of recovering data from non-zeroed operating system pages, page files, memory object caches [7], [23] and non-zeroed memory from other security domains. Likewise, Halderman explored memory remanence for encryption keys [17] and Chan explored the security implications of preserving memory contents [8].

In contrast to static approaches, Lin [19] explores live techniques to extract and reverse data structures from execution. Cozzie [13] takes a different approach by applying Bayesian machine learning to classify unknown data structures.

Furthermore Chen [10] and Burzstein [5] have explored how private browsing modes work and how residue objects may limit their effectiveness.

III. Design

A. Valgrind

Cafegrind is designed as an extension to Valgrind [21], a suite of tools for debugging and profiling. Common functionalities of Valgrind include memory leak detection, cache simulation and program analysis to detect concurrency bugs. Valgrind executes target programs by dynamically translating them into its internal VEX execution format. As a result, Valgrind is able to perform fine-grained instrumentation at the lowest levels of the machine's architecture. Unlike similar emulation systems such as QEMU, Valgrind is also able to interpret higher-level debugging symbol information to support various functionalities such as memory leak detection. Cafegrind builds upon these intrinsic features to track the life-cycle of data and provides additional insight into specific data structures by performing automatic type inferencing.

B. The Life-cycle of Data

The life-cycle of data in a program is shown in Figure 1. First, memory is allocated by using a function such as malloc() or `new` and it is then initialized by a function such as memset(), C++ constructor or memory pool constructor. Once the base object is ready, its fields are populated with information and the data structure is accessed and modified as the program runs. Once the data structure is no longer needed, it is freed and its memory returns to a pool for reallocation. Throughout this process, memory locations can be overwritten by modification, initialization and reallocation. However, the process of relinquishing memory does not always clear the latent contents of the data structure. In many cases, data is only partially destroyed as reuse of a memory area does not always completely overwrite old data. This partial destruction process is one of the underlying principles behind volatile memory forensic analysis and is useful in uncovering freed data. Cafegrind uses empirical methods to track how much data can be recovered from memory dumps that contain both active and freed data.

C. Type Inferencing

Since C/C++ are not strongly typed languages, Cafegrind must infer the type of allocated memory areas to build its object map. To illustrate how this works, consider the following code snippet:

```
[1] struct datastructure * mydata;
[2] mydata = (struct datastructure *)
malloc( sizeof( struct datastructure ) );
```

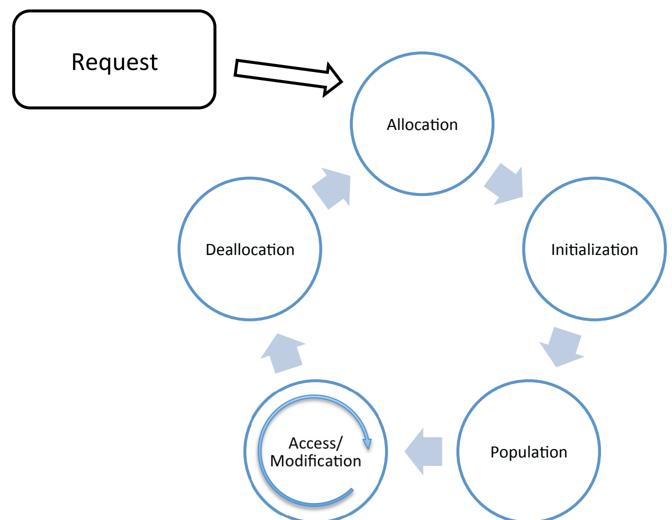


Fig. 1. The lifecycle of data

```
[3] mydata->fld1 = 100;
```

In the above example, on line 1, the program creates a pointer to an object of type `datastructure` on the stack. Line 2 calls malloc() to allocate memory for type `datastructure`, but malloc() returns a memory area with the type void* which is then cast back to the proper type. Since this type casting is not present in the assembly code, we have to rely on an alternate method to infer the type of the memory allocation. The important assignment here is on line 2. When Cafegrind observes that the malloc memory area is assigned to a local stack variable, Cafegrind propagates the type information from the stack variable to the memory area. In assembly code, line 2 is implemented by using a store instruction where the target of the store is the address of the stack variable `mydata` and the value written to it is the address returned by malloc(). Cafegrind first checks if the type for the target address is known and having determined that the type is `struct datastructure` we can propagate the type to the memory area described by the value in the instruction. Cafegrind uses this incremental process to build and maintain type information for dynamic memory allocations.

Advanced inferencing techniques
Complete type inferencing is a challenging task because of the plethora of ways in which programs interact with loosely typed data. We briefly discuss some strategies to achieve more complete coverage. Some of these strategies are experimental and we are working on evaluating the benefits of these strategies.

First, precise member-level tracking is required for accurately tracking the type of structures. In some cases, local stack variables could point into nested objects within an existing allocation or into a generic array and we keep track of the offsets of constituent members to increase the accuracy. Secondly, there could be instances where a heap allocation is directly assigned to another heap allocation. For instance, if a structure has a pointer to a buffer that is allocated on demand, the memory area provided by malloc() may be directly assigned to a pointer belonging to a dynamic allocation on the heap. In this case, we use any existing inference we have about the source of the assignment and propagate the appropriate type to the destination.

Alternatively in C++, types can be inferred by monitoring object constructors. When *new* is invoked, all constructors in an object's inheritance hierarchy are called starting from the most generic to the most specific. For instance, a *square* object may run its constructors in the following order: shape→rectangle→square. In this case, the object type inference would be made when the square constructor is called with a pointer to a square implicitly passed with *this* argument. Since C++ stores its constructors in a special ELF [1] section called **.ctors**, we can adequately identify which functions serve as constructors for type inferencing. Another safety check is to ensure that the name of the constructor is consistent with the purported type of the object.

Finally, ambiguous types involving unions or generic arrays cannot currently be resolved using our framework. To solve this problem, we would need to develop a type agreement and type history system. For instance, a generic array may initially be recognized as a char* array at first, but then in a different context, the type resolution system may identify that various offsets in the array correspond to ethernet packets. The system should recognize and promote subtypes as necessary in order to maintain high fidelity type inferencing.

D. Methodology

In order to track the data life-cycle of a program and also build an object map by performing type inferences, Cafegrind instruments the following events while a program is running:

1) Memory allocation
2) Data structure read/write accesses
3) Memory deallocation

To support type inferencing, Cafegrind intercepts all memory allocation and deallocation requests. When an allocation is made, Cafegrind tracks the memory object returned by malloc() and the stack trace at the time of the allocation. Recall that malloc() does not return typed objects; objects are of the generic type void*. These allocations are stored in the object map, which is implemented as an efficient ordered set for fast lookup and retrieval. Cafegrind only discovers the type of a memory object when the object is loaded into a typed pointer object as described in Section III-C. This is enabled by intercepting store instructions to memory locations and Cafegrind can thus track the assembly level assignment of a pointer to the memory object to a stack location. If the pointer points to a tracked memory location, Cafegrind performs a lookup on the debug information associated with the executable to describe the type of the stack variable. This process queries the debug information present in loaded libraries and binaries in the DWARF3 [2] format and helps identify the type of the stack object. Once the type of the stack object has been identified, it is then propagated to the dynamically allocated memory object and stored in the same ordered set.

In addition to performing type inferencing, Cafegrind also tracks accesses and modifications to the data structure. This allows analysis of the access patterns to be associated with a particular type. These accesses are tracked by instrumenting all memory loads and stores. In the previous code snippet, line 3 illustrates how Cafegrind tracks data accesses. When a memory address is accessed, Cafegrind checks its internal allocation database to see whether or not the access belongs to a tracked allocation. If so, Cafegrind identifies which allocation the access belongs to and resolves the member being accessed. These accesses are tracked and aggregated statistically to reveal how the underlying data types are being used. Furthermore, we also track which function call led to the data access and aggregate this information to find which data types a particular function accesses. Cafegrind once again uses efficient algorithms based on ordered sets to perform the lookup and updates quickly.

Cafegrind also maintains a set for allocations that have been freed by the application. This set is used to track when objects are overwritten in memory and helps determine the time at which data is destroyed. In addition to maintaining properties of data structures, we also collect their binary contents. This is useful for off-line analysis using utilities like `strings` and the contents are clustered by their type in separate files to enable easier processing.

By performing such monitoring, Cafegrind is able to better understand how data is created, accessed and destroyed. This can provide an empirical analysis on which data structures could be found and for how long they are expected to persist. For instance, a forensic analyst may find some interesting information in an HTML cache object, but this type of object may be transient and the data stored in it can change throughout a browsing session as the user visits certain websites. Cafegrind can provide an analysis of the longevity of such data.

IV. Evaluation

A. Experimental Setup

Our technique described in Section III relies on explicitly monitoring canonical variable accesses and assignments. Common compiler optimizations can store pointers in registers instead of allocating space on the stack. This process can affect Cafegrind's type inferencing algorithm, thus Cafegrind only works on binaries and shared libraries that are compiled with debugging options enabled and optimization disabled. From a forensic standpoint, an investigator is unlikely to encounter a machine with such a configuration. However, the purpose of this paper is to study the ideal behavior of data structures and applying these alterations does not operationally affect the behavior of the applications we study except for imposing additional runtime overhead when the program is being instrumented.

Our experiments were run on a single Intel Core i7 920 CPU running at 2.67 GHz with 6GB of RAM running Gentoo Linux 1.12.14 on the 2.6.34-r12 kernel. All the applications and libraries were compiled with debugging enabled and optimization disabled. This configuration affords Cafegrind visibility into the inner workings of system applications and libraries. As a result, we can trace a complete execution chain of all the subsystems such as the KDE desktop environment if desired.

B. Basic Concepts

In this paper, we focus on web browsers because of their increasing popularity and importance. We study Firefox and Konqueror, two open source web browsers and measure the effectiveness of "private browsing mode" against core dump analysis. Further, we also look at which data structures could potentially leak private information and also study the similarities and differences between the two browsers.

For each object in an application, we track the following attributes:

TABLE I
COVERAGE

Application	Store Coverage	Load Coverage	Overall Coverage
Firefox	70.48 %	88.11 %	83.51 %
KWrite	85.8 %	94.27 %	92.66 %
Links	99.09 %	99.99 %	99.6 %
Tor	85.95 %	96.43 %	95.02%

1) Type - The type of an object
2) Object Size - The size of an object
3) Age - The length of time an allocation lasts before it is deallocated
4) FreedAge - The length of time a deallocated structure lasts before it is clobbered by a subsequent allocation
5) Reads - Number of reads performed
6) Writes - Number of writes performed
7) Allocation Size - Size of the allocation including slack

Our evaluation methodology cross-analyzes these attributes to find correlations which reveal patterns and relationships in the way that these structures are allocated, accessed and freed. We measured forensically interesting data structures in several ways. First, we apply well-known string identification algorithms to find ASCII strings in memory. This helps us identify web pages and XML documents as well as HTTP requests that are latent in memory. Secondly, we perform outlier analysis [4] on the dataset to find objects that are large, frequently accessed, or have great longevity. Outlier analysis is useful to find data structures which have different properties based on configuration changes. Cafegrind thus produces a list of candidate types that the forensic analyst can inspect more closely to find evidence of interest. In this paper, we have taken the candidate list and applied our expertise to identify and measure the characteristics of interesting types of forensic evidence.

C. Coverage

The type inferencing coverage is measured by counting the number of load/store operations on dynamically allocated regions of memory for which Cafegrind has inferred the type. In terms of type inferencing coverage, Cafegrind performs remarkably well. As shown in Table I, we have seen upwards of 90% type inferencing accuracy for several real world applications. This behavior should be expected since the type information comes from debug information ingrained in the program itself. The coverage was also improved as the system that was used for evaluation contained shared libraries built with debug information. Additionally since we tracked load/store instructions from the program and its linked libraries, our type inferencing technique was able to get a clear insight into the creation and usage of data types. However we have not compared the accuracy of the inferred types with the ground truth due to the complexity of obtaining the ground truth for large applications and there are several factors that can affect the performance of the type inferencing system:

1) Generic arrays of char* and similar types that are cast into specific types before access.
2) Variable length structures where the last element is a void* or char*.
3) Unions used in structures where the type is ambiguous.
4) Nested types with unions. Cafegrind currently doesn't handle nested type agreement.
5) The use of custom allocators which return void*. Currently, Cafegrind isn't able to follow multiple levels of type propagation.

Nonetheless, from experience and cross-validation we find that Cafegrind performs very well in real-world scenarios. Certainly, there are cases where programs use certain practices that may be challenging for type inferencing systems to handle. As we have seen with Firefox there are large code-bases that use pointer wrappers, templating and other constructs which can obscure the true type of an object. This remains a challenge for even the best type inferencing systems. In the absence of compiler assistance, there are not many options to handle these cases. For example, KOP [6] relies on additional compiler information to extract the type assignments.

D. Applications

Firefox

Firefox is a popular web browser that supports private browsing mode. When Firefox starts up, it allocates several singleton UI and bookkeeping structures. As the user opens web pages, Firefox creates a number of HTML parsers, XML parsers, UI widgets and graphical image renderers for PNG/JPG images present on a webpage. The purpose of our study is to measure how many of these elements are created when using the private browsing mode and are latent in memory for an extended period of time. This measurement provides a rough measure on how much forensic evidence is available during core dump analysis. We also identify which data structures contain sensitive information .

Figure 2 shows a histogram of the distribution of object ages. Many of the objects allocated by Firefox have a long lifespan. This is likely to be the case because Firefox uses a custom allocator and smart pointers.

Figure 3 shows how long freed objects last in memory before they are ultimately reallocated and clobbered. There seem to be three distinct clusters representing long-term, medium-term and short-term reallocations. This behavior is reflective of how the memory allocator redistributes memory. Smaller allocations are more frequent and therefore, the longevity of their data is also shorter because these smaller memory pools are heavily used. Larger allocations tend to be more rare and thus latent data has a longer life expectancy in these pools. However, if the system is running low in memory, larger pools can be split and reallocated to service requests for smaller allocations.

These results confirm our observations about how Doug Lea's malloc() allocator [18] is implemented in GLIBC 2.x. This allocator tends to have the following properties(documented in the source code): small allocations are made from a pool of quickly recycled chunks, large allocations (>= 512 bytes) are made in FIFO order and very large allocations (>= 128 KB) are made using system memory facilities. Much of the evidence we found was stored in large ring buffers or cache structures which tend to be allocated in larger pools and since larger allocations have a longer life expectancy, we believe that volatile memory forensics can be used to extract useful information from applications.

We perform a conjoint measurement of Firefox to better characterize how private mode manages data structures. Since private mode cannot be used in isolation, we run a series of conjoint actions depicted in Figure 4 as an experiment meant to isolate the effects of private mode.

Figure 4 describes the methodology we used during our experiments. We denoted with '[F,W]' the action of launching Firefox in Normal Mode and visiting a known webpage. We denoted with '[F,P,W]' the action of launching Firefox in Normal mode, activating the Private mode option and then visiting

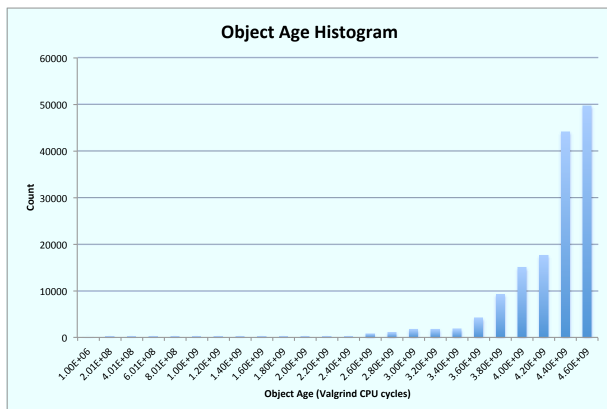


Fig. 2. Firefox: Object age histogram

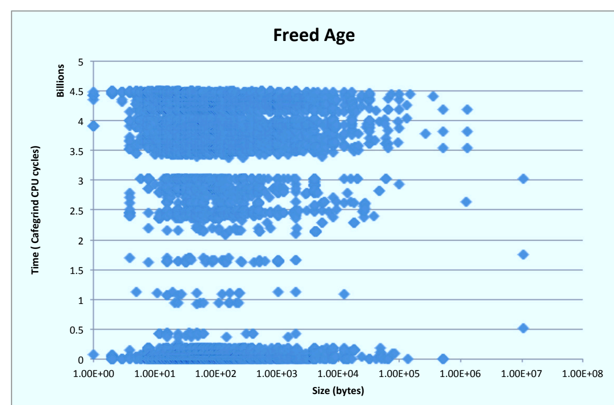


Fig. 3. Firefox: Freed object age vs size

the same webpage we visited in '[F,W]'. As part of another task, denoted by '[F,N,W]', we proceeded to launch Firefox in Normal Mode, opening a tab and then visiting our test webpage. We also validated our experiments by doing three more tests involving enabling the Private Mode in between launching the Firefox program, opening a new tab and actually visiting the webpage, but results from these experiments did not have any significant differences from the ones presented in the paper. Another task we included was launching Konqueror and visiting the page we previously visited with Firefox. For all our tasks we collected in the background all the information using Cafegrind. Our results are shown in different sections of Figure 5. Figure 5a shows how Firefox in Normal mode accesses various data structures and how Private Mode differs by accessing different data structures. Figure 5b presents the differences between accessing Firefox in Normal Mode and Firefox New Tab. Initially we suspected that Firefox in Private mode accessed unique types of data structures, but when comparing Figure 5a against Figure 5b we noticed there existed some overlap. Upon a deeper analysis we refined our list of unique data structure types that Firefox in Private Mode uses which we depict in Table 5d. By analyzing the Firefox source code, we were able to verify that private mode uses separate state storage objects for browser history and DOM structures and this is reflected in the new instances of storage and file streams accessed. Moreover, we noticed that every private mode instance tab has a separate Javascript stack context.

In Figure 5c, tables 5e and 5f we present how Firefox and Konqueror access different types of data structures. We observed a significant overlap between the types of data structure accessed by each of the program. We found that these datastructures were from shared libraries such as fontconfig and X11. We also noticed that there were some datastructures which were only used in either Firefox or Konqueror. By analyzing Cafegrind output we found these to be related to the fact that they use different HTML rendering engines.

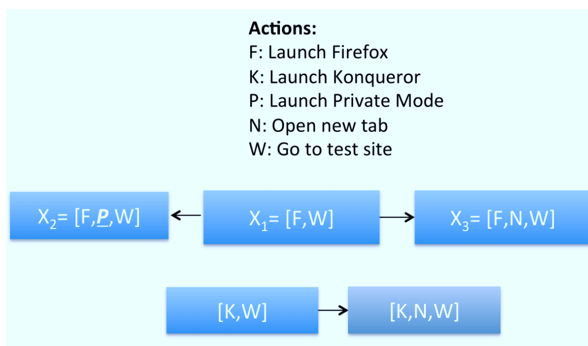


Fig. 4. Conjoint measurement of data structures

Furthermore, we ran Cafegrind on Firefox and determine which data structures retain information from the private browsing mode. We launched Firefox, entered private browsing mode, visited a popular tech website and stopped private browsing mode. At this point, we enabled the dumping of accessed/freed memory structures with more than 40% ASCII characters and proceeded to close Firefox. In searching these dumps, we found a plethora of information left over from the private browsing session, some of which is shown below.

Private Browsing: Residual Data in Firefox:

1) GStringr, gconvinfosteps, nsAttrValuemBits, nsCAutoString, nsCOMPtrnsIContent, nsCOMPtrnsIURI, nsEntryheader, pngstructdefjmpbuf - All contained the URL as well as a variety of data from the visited website
2) Tokenz - Contained SQL statement intended to clean up private browsing mode
3) nsXPTCVariant - Contained a large assortment of data, ranging from PNG files to URLs to various private browsing resources, such as cookies
4) JSHashEntrynext - Contains a variety of URLs, many linked to javascript and XML files
5) ScopedXPCOMStartupmServiceManager - Cache contains a variety of information, including URLs and SQL statements

Outside of private browsing mode, we found the following structures to be key to the operation of Firefox. The list is extensive, so we present an abbreviated list of the most important types here.

Interesting Structures Include:

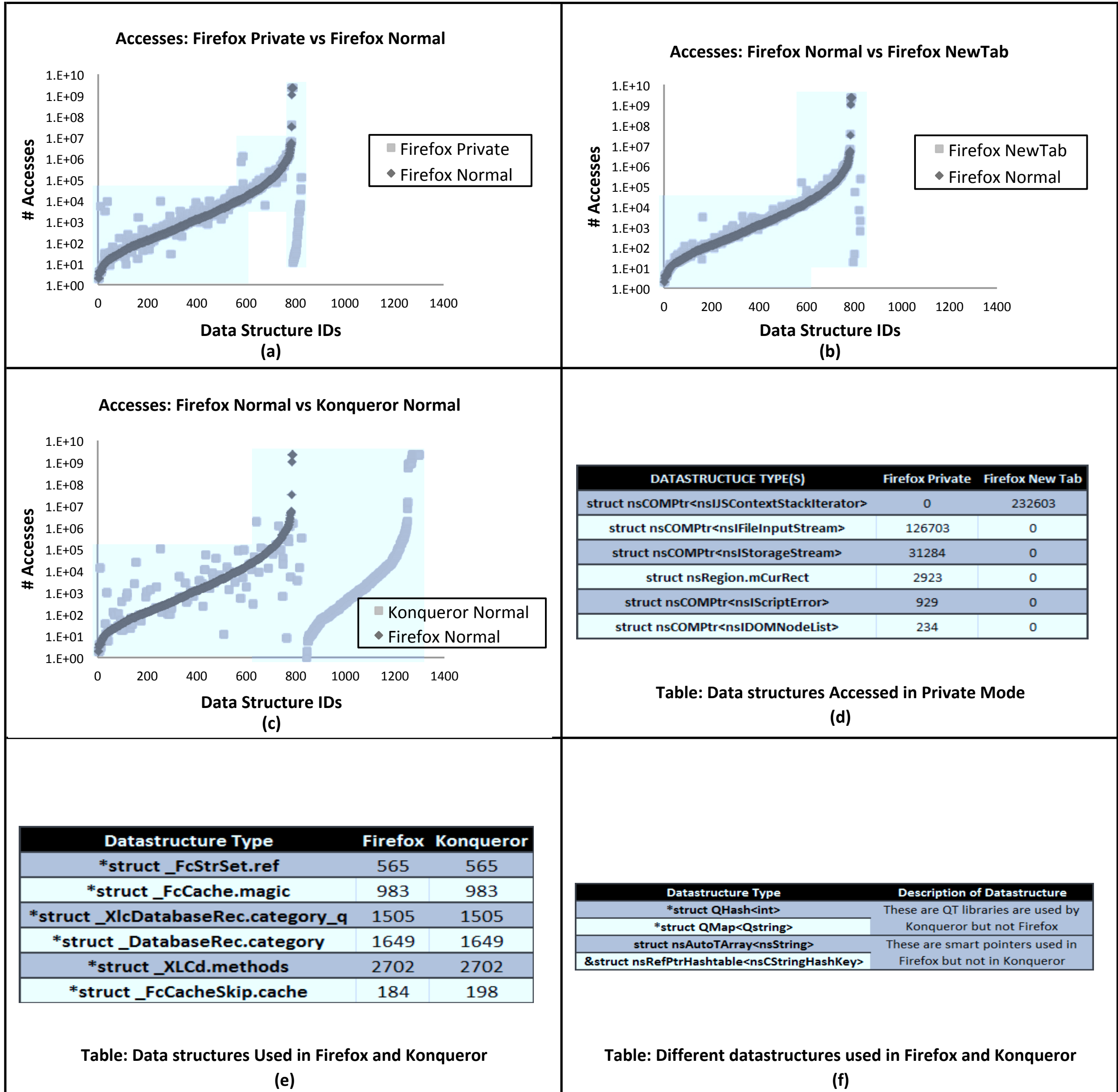
1) GCGraphBuilder - Not surprisingly, the garbage collector's data structure is the most frequently written object. It has to track other objects for garbage collection and has a long lifetime.
2) nsCOMPtr$< nsICSSParser >$ - This is an instance of a CSS parser. It has a huge number of writes. nsCOMPtr is a smart pointer that manages memory to prevent leaks and has a long lifetime.
3) XML_ParserStruct - Parses XML structures and has a relatively short lifetime.

E. Code Metrics

Cafegrind is over 2,100 lines of code and it builds upon the Valgrind framework which has over 87,000 lines of code. Adding tracing and instrumentation functionality in Cafegrind is relatively straightforward by using standard Valgrind functions.

F. Performance

We compared the performance of Cafegrind against the performance of the Valgrind tool itself in Table II. Valgrind in its purest form imposes a modest performance penalty because it does binary translation and intercepts function calls. Lackey



DATASTRUCTUCE TYPE(S)	Firefox Private	Firefox New Tab
struct nsCOMPtr<nsIJSContextStackIterator>	0	232603
struct nsCOMPtr<nsIFileInputStream>	126703	0
struct nsCOMPtr<nsIStorageStream>	31284	0
struct nsRegion.mCurRect	2923	0
struct nsCOMPtr<nsIScriptError>	929	0
struct nsCOMPtr<nsIDOMNodeList>	234	0

Table: Data structures Accessed in Private Mode
(d)

Datastructure Type	Firefox	Konqueror
*struct _FcStrSet.ref	565	565
*struct _FcCache.magic	983	983
*struct _XlcDatabaseRec.category_q	1505	1505
*struct _DatabaseRec.category	1649	1649
*struct _XLCd.methods	2702	2702
*struct _FcCacheSkip.cache	184	198

Table: Data structures Used in Firefox and Konqueror
(e)

Datastructure Type	Description of Datastructure
*struct QHash<int>	These are QT libraries are used by Konqueror but not Firefox
*struct QMap<Qstring>	
struct nsAutoTArray<nsString>	These are smart pointers used in Firefox but not in Konqueror
&struct nsRefPtrHashtable<nsCStringHashKey>	

Table: Different datastructures used in Firefox and Konqueror
(f)

Fig. 5. Firefox and Konqueror analysis

TABLE II
PERFORMANCE

Application	Time
Firefox native	6 s
Firefox Cafegrind	3:30 m
Firefox Lackey	2:57 m
Firefox Memcheck	1:52 m
Konqueror	2 s
Konqueror Lackey	3:12 m
Konqueror Cafegrind	3:26 m

[21] is an example tool in Valgrind which traces memory accesses in addition to other instrumentation. When Lackey is used, the performance quickly degrades as each instruction is individually executed with memory tracing support enabled. Since Cafegrind uses facilities that are similar to what Lackey does, its performance is similar to that of Lackey. Further performance profiling showed that the additional functionalities of type inferencing and object tracking contribute around 29% and 7% overhead respectively.

V. DISCUSSION

Volatile memory forensics is still a nascent field in many ways. Current techniques developed to extract evidence of interest often rely on expert knowledge or some intuition about the structure of evidence. Current approaches have explored extracting ASCII data and data of high entropy. We believe that these approaches can be complemented by the use of statistical data to further identify these structures. Cafegrind is a step in this direction because it helps to assess the practicality of data extraction and automatically identify target types.

In the long term, we believe that the collection of programs that need to be considered in a forensic investigation can be quite large and better systematic approaches to forensics are necessary to address the natural diversity in software systems. In the process of doing so, it is absolutely necessary to establish a ground truth as a baseline to measure evidence extraction against. Since the majority of popular applications are written in a loosely-typed language, it becomes necessary to adopt type-inferencing and type-discovery methods to effectively capture an accurate object map.

Another trend that affects analysis is the use of modular components in software. For instance, many applications embed web browsers, movie players and use common encryption libraries. Learning to recognize forensic evidence in one instance of a library is sufficient, because the same techniques can be equivalently applied to all other similar instances. However, this level of sharing illustrates how complicated the software stack can be. In the course of our analysis, we found that some applications such as web browsers use many shared libraries and in some cases such as Konqueror, the original application binary simply launches an instance of a shared web rendering framework called WebKit.

Furthermore, since libraries have strict function export interfaces and well-defined data structures for interaction, crossing library boundaries can reveal a wealth of forensic information. This is not surprising because these interfaces have been used as type-revealing operations in type-sink systems [19]. Likewise, the use of these libraries often requires data type conversion between different kinds of data structures and this results in duplication of data. This duplication happens at the data structure level where the structures may be shadowed in different libraries or in buffers where libraries pass information to each other. For instance, a web browser might use *libxml* to parse an XML file and then use *libqt* to display it. An XML file parsed in this workflow would appear in the private data of both libraries. Additionally, data can be buffered when it is being compressed or encrypted. The original buffer contains a copy of the data as well as the compression/encryption library's temporary buffer. As a result, there are many copies of same data present in memory and our analysis is just the first step in shedding some light on the associations between these structures.

Attribution is another important issue in this discussion. Some data structures found in web browsers may not have been directly created as a direct consequence of user actions. For instance, an advertisement served on a page may not have been knowingly requested by the user and any forensic analysis should take the source of such evidence into account. This effect is especially evident with Tor where anonymized network packets may be routed through client nodes. Any forensic evidence collected from Tor should take into account the source of the data with proper attribution. Further work needs to be done on this attribution process.

VI. CONCLUSION

Forensic analysis of evidence gathered from volatile memory is a nascent but important field. Advances in algorithms and methodologies supporting this extraction process seem to be headed toward a better understanding of the semantics and contents of this information. To help assess the practicality of extracting evidence from these memory dumps, we attempt to establish a baseline for the object map by using the algorithms described in this paper. Although the initial results are encouraging, much work still remains to make full evidence extraction from volatile memory dumps a reality.

Several important challenges remain to be solved. First is the problem of type classification given the binary contents of an object. Cozzie [13] was a good step, but we found that this problem is perhaps more challenging than it initially appears. For instance, even though we are monitoring seven attributes, unique identification of a type requires additional information. On the flip side, these attributes may be able to help guide a forensic analyst toward interesting evidence by virtue of identifying the characteristics of relevant data.

Looking forward, we expect that advances in leveraging peripheral information from compilers and debugging information will help in the identification process. Our work is just one step to drive this process forward by identifying and characterizing such data.

Acknowledgements We would like to thank the anonymous reviewers for their valuable feedback. This research was supported by grants from DOE DE-OE0000097 under TCIPG (tcipg.org) and a Siebel Fellowship. The opinions expressed in this paper are those of the authors alone.

REFERENCES

[1] Executable and Linking Format. http://refspecs.freestandards.org/elf, 1995.
[2] DWARF 3.0 Standard. http://dwarfstd.org/Dwarf3Std.php, 2005.
[3] K. Amari. Techniques and Tools for Recovering and Analyzing Data from Volatile Memory, 2009.
[4] V. Barnett and T. Lewis. *Outliers in statistical data*. Wiley series in probability and mathematical statistics: Applied probability and statistics. Wiley & Sons, 1994.
[5] G. Burzstein, C. Jackson, and D. Boneh. An Analysis of Private Browsing Modes in Modern Browsers. In *Proceedings of the 19th USENIX Security Symposium*. USENIX Association, 2010.
[6] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS 2009)*, pages 555–565, Chicago, Illinois, USA, 2009. ACM.
[7] A. Case, L. Marziale, C. Neckar, and G. Richard III. Treasure and tragedy in kmem_cache mining for live forensics investigation. *Digital Investigation*, 7:S41–S47, 2010.
[8] E. Chan, J. Carlyle, F. David, R. Farivar, and R. Campbell. BootJacker: Compromising Computers using Forced Restarts. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 555–564. ACM, 2008.
[9] E. Chan, S. Venkataraman, F. David, A. Chaugule, and R. Campbell. Forenscope: A framework for volatile memory forensics. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010)*, Austin, TX, USA, 2010.
[10] S. Chen, H. Chen, and M. Caballero. Residue objects: a challenge to web browser security. In *Proceedings of the 5th European conference on Computer systems*, pages 279–292. ACM, 2010.
[11] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium*, page 22. USENIX Association, 2004.
[12] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*, page 22. USENIX Association, 2005.
[13] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging for Data Structures. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
[14] U. Drepper. What every programmer should know about memory. http://people.redhat.com/drepper/cpumemory.pdf, 2007.
[15] S. Garfinkel. Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4:2–12, 2007.
[16] P. Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings of the 10th USENIX Security Symposium*, pages 39–54, Washington, D.C., 2001. USENIX Association.
[17] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, and J. A. Calandrino. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.
[18] D. Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 2000.
[19] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS10), San Diego, CA*, 2010.
[20] Microsoft. Kb 244139: Windows feature allows a memory dump file to be generated with the keyboard. http://support.microsoft.com/kb/244139/en-us.
[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2007)*, pages 89–100, San Diego, California, USA, 2007.
[22] A. Savoldi and P. Gubian. Blurriness in Live Forensics: An Introduction. In *Proceedings of Advances in Information Security and Its Application: Third International Conference, Seoul, Korea*, page 119. Springer, 2009.
[23] A. Schuster. Searching for Processes and Threads in Microsoft Windows Memory Dumps. The Proceedings of the 6th Annual Digital Forensics Research Workshop, 2006.
[24] Volatile Systems. Volatility Framework. http://www.volatilesystems.com/VolatileWeb/volatility.gsp.