Poster: Post-Intrusion Memory Forensics Analysis

Pengfei Sun
Electrical and Computer Engineering
Rutgers University
pengfei.sun@rutgers.edu

Rui Han
Electrical and Computer Engineering
University of Miami
r.han@umiami.edu

Saman Zonouz
Electrical and Computer Engineering
Rutgers University
saman.zonouz@rutgers.edu

***Abstract*—A yet-to-be-solved but very vital problem in forensics analysis is accurate memory dump data type reverse engineering where the target process is not a priori specified and could be any of the running processes within the system. We present a lightweight system-wide solution that extracts data type information from the memory dump without its past execution traces. Our proposed solution constructs the dump's accurate data structure layout through collection of statistical information about possible *past* traces, forensics inspection of the *present* memory dump, and speculative investigation of potential *future* executions of the suspended process. First, the engine analyzes a heavily instrumented set of execution paths of the same executable that end in the same state of the memory dump (the eip and call stack), and collects statistical information the *potential* data structure instances on the captured dump. Second, the engine uses the statistical information and performs a word-by-word data type forensics inspection of the captured memory dump. Finally, the engine revives the dump's execution and explores its potential future execution paths symbolically. It traces the executions including library/system calls for their known argument/return data types, and performs backward taint analysis to mark the dump bytes with relevant data type information. Our solution's preliminary experimental results are very promising (98.1%), and show that it improves the accuracy of the past trace-free memory forensics solutions significantly while maintaining a negligible runtime performance overhead (1.8%).**

I. Introduction

Software reverse engineering has been a challenging and recurring problem in computer security that aims at recovery of high-level program abstractions [1]–[4]. The results could potentially be used for various purposes such as memory forensics, malware development and analysis, reversing cryptographic algorithms and network protocols, digital right management, and auditing program binaries. Specifically, a desirable capability in many of those security and forensics applications is automatic reverse engineering of data structures given only the memory dump of a process and without the execution trace information.

The existing data type reverse engineering solutions are categorized into three groups. First, static binary executable analysis techniques extract data structures defined within an executable through disassembly [5] or symbolic execution [6]. Second, dynamic execution analysis solutions [5], [7] trace the execution to reverse engineer data types using type-revealing instructions [7]. Third, static memory analysis techniques perform forensics directly on memory dumps for data structures.

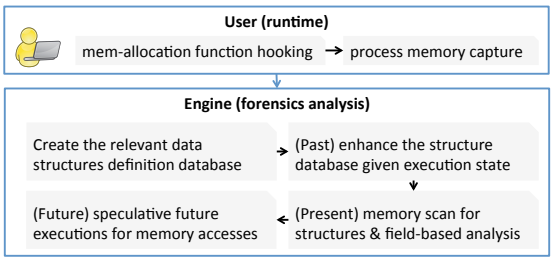

Fig. 1: The High-Level Architecture

Laika [8] sweeps the memory for pointers and assumes all pointers point at other data structures.

The past work falls short for practical forensics analysis of applications' memory dumps: *i)* static executable analyses can reverse engineer executable-defined data structure definitions accurately; however, those approaches by themselves are of limited use for memory forensics when the execution trace is not available; *ii)* dynamic execution monitors cause a very high performance overhead $> 6X$ on the target process. Hence, it is infeasible to trace all the running processes on a system as any of them may be misbehaving and needing forensics analysis; and *iii)* the static memory analyses are not sufficiently accurate in practice (e.g., 70% for Laika [8]).

We present a hybrid memory forensics solution that leverages the high accuracy of static executable analyses (*i*) and dynamic execution monitoring (*ii*) to provide a low overhead and precise static memory dump forensics (*iii*) when the execution trace is not available.

Our contributions are thus the following:

- We introduce a trace-free memory data structure forensics solution with high reverse engineering accuracy and negligible 1.8% runtime overhead.
- We present a probabilistic information fusion method to combine prior statistical information about possible past traces, results from the present dump forensics and speculative investigation of potential future executions of the suspended process.
- We present a preliminary evaluation on real-world settings, i.e., CoreUtils suite.

II. Design Overview

Figure 1 shows our solution's high-level architecture that contains a forensics analysis framework to where the user uploads the target process'es memory image and the application executable. The process starts by the engine's installation on

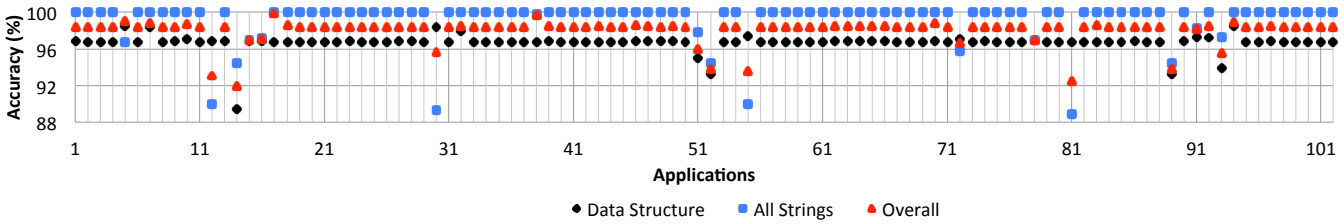

Fig. 2: Memory Data Type Reverse Engineering Accuracy

the user's machine. Upon detection of a suspicious running process by the user or an intrusion detector, the engine's client suspends the process and uploads its memory image for forensics analysis.

The engine performs its forensics analysis through the following three stages: *past*, *present*, and *future*.

Past. The engine leverages offline analysis to maximize its runtime analyses' efficiency once a particular running process is picked for analysis. To create the data structure definition database, we implement automated static code analysis modules to investigate available library and kernel sources searching for data structure definitions. There often exists different data structures with identical names defined at different points across the system. The engine maintains the context where each data structures is defined. The engine explores and analyzes the memory dump (memdump)'s possible past execution trace. It starts from the executable's entry point, and statically explores the call and control flow graphs for possible paths to the captured dump's execution state (the instruction pointer and the call stack). Using symbolic execution, the engine filters out the infeasible paths and generates the corresponding test cases. Through an instrumented execution of the test cases, the engine logs the structure memory allocations and collects statistical information about the potential structures on the captured dump. For instance, if a particular data structure exists in almost all test case executions, it exists in the captured dump with higher probability than a data structure that was never encountered during the test case executions.

Present. The engine performs a forensics analysis of the captured dump using the created models (data structure definition database and statistical information). It sweeps the dump for landmark signatures (inserted by the hooked API), and extracts every structure's base address and size. The engine marks every memory address with possible data types using the memory value and the engine's forensics rules. The engine uses its forensics results and the created models to calculates a ranked list of the best matching data structures for each memory location.

Future. The engine revives the captured dump's execution, explores all feasible future branches of the code symbolically to generate test-cases. It runs the executable with the test cases, while it reverse engineers data type revealing instructions and the library/system calls with known return/argument data types. The engine implements backward data taint analysis on each test-case trace to backtrack the revealed data types to a memory address of the captured dump.

III. Evaluations

We evaluated our solution on CoreUtils v8.22. We suspended each process at a random execution point, i.e., half way through its finish time. The user desktop ran Linux kernel v3.11.

We measured the accuracy of our solution's ultimate data type forensics. Figure 2 shows the results for strings and other data structures of each application separately. Our solution correctly recognized 99.2% of the strings and 96.7% of the memdump data structures correctly. Its overall accuracy level 98.1% is very promising even though our solution did not have access to the execution traces before the memory capture point.

IV. Conclusions

We presented a hybrid data structure reverse engineering solution that takes the memory image for a selected running process on the user's machine, and determines its semantic data structure layout without the need for execution traces before the memory capture point. The engine performs a static forensics analysis of the captured memory dump, and its potential past and future execution traces. Our solution correctly reverses engineers 98.1% of the data structures in real-world application dumps with 1.8% runtime performance overhead.

Acknowledgments

We appreciate the Office of Naval Research for their support of our project under the project No. N00014-15-1-2741.

References

[1] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Information systems security*. Springer, 2008, pp. 1–25.

[2] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: a binary analysis platform," in *Computer aided verification*. Springer, 2011, pp. 463–469.

[3] E. J. Chikofsky, J. H. Cross *et al.*, "Reverse engineering and design recovery: A taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13–17, 1990.

[4] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Vcr: App-agnostic recovery of photographic evidence from android device memory images," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 146–157.

[5] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs." in *NDSS*, 2011.

[6] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS*, 2011.

[7] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," 2010.

[8] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures." in *OSDI*, vol. 8, 2008, pp. 255–266.