Atlantis: Improving the Analysis and Visualization of Large Assembly Execution Traces

Huihui Huang, Eric Verbeek,
Daniel German, Margaret-Anne Storey
University of Victoria
Victoria, BC, Canada
Email: norah, everbeek, dmg, mastorey@uvic.ca

Martin Salois
Defence Research and Development Canada - Valcartier
Quebec, QC, Canada
Email: martin.salois@drdc-rddc.gc.ca

***Abstract*—Assembly execution trace analysis is an effective approach to discover potential software vulnerabilities. However, this analysis is labour intensive due to the lack of source code and the huge size of the execution traces. Instead of browsing billions of instructions one by one, software security analysts need higher level information that can provide an overview of the execution of the program to assist in the identification of patterns of interest in the executed program. The tool we are presenting in this paper has a number of features that make it particularly successful in achieving this goal. These features are improvements we have made to Atlantis, our trace analysis environment for multi-gigabyte long assembly traces. The contributions of this continuous work falls into three main categories: a) the ability to efficiently reconstruct and navigate the memory state of the program at any point in the trace; b) the reconstruction and navigation of functions and processes, and c) a powerful search facility to query and navigate the trace. Software nowadays is increasingly complex and many applications are designed as collaborative systems or modules interacting with each other, as a result making the discovery of vulnerabilities extremely difficult. The contributions of this paper extend the security analyst's capability to investigate vulnerabilities of real-world large execution traces, and can lay the groundwork for supporting trace analysis of interacting programs in the future.**

I. Introduction and Background

Software vulnerabilities can compromise a computer or even an entire internal network, exposing data and control systems to attackers [1]. While software companies are expected to prioritize creating secure software and invest a significant amount of resources in the process as a first line of defense, it is often the case that software is shipped with vulnerabilities that can expose it to attacks [2].

A second line of defense is to perform software auditing. Auditors attempt to find vulnerabilities in these systems by testing and studying them. However, these Auditors often lack access to source code, creating additional technical challenges for an already-difficult activity [3].

Dynamic analysis [4]—which records and analyses the assembly execution trace of a running program—is one of the main methods used in vulnerability detection. Traces contain every micro instruction executed by the program, often billions of them, resulting in large and opaque traces. Without access to the more readable and concise source code, analysing these trace is a labour intensive task.

As part of our ongoing research work, we have previously developed Atlantis, an assembly trace analysis environment designed specifically to work with execution traces generated for security analysis. It was designed as a user-friendly environment to provide security engineers with the ability to inspect and navigate large traces. Users are able to browse the trace, add comments, and mark regions of interest (this version of Atlantis was demonstrated at WCRE 2012 [5]).

This paper showcases the contributions we made for assembly execution trace analysis by implementing several novel and powerful features in Atlantis. As we know, with the newly implemented features, Atlantis is the only tool that can reconstruct the memory state of the trace as well as visualize the function call pattern and process and executable interaction patterns. All of these unique and novel features allow software security analysts to perform analyses with new insights and allow scaling to very large traces.

In software security analysis, there is an assumption that all memory corruption vulnerabilities should be treated as exploitable until you can prove otherwise [3]. To determine a program's potential exploitability from a trace, it is important to know, at any point in the trace, the state of the memory used by the program. That is at any instruction, what memory has the program accessed, and what are its current contents. This feature enables analysts to observe how the program accesses and changes its memory in order to find out the misuse of memory.

However, implementing this feature is not trivial. Cleary et al. proposed several methods to solve this problem [6], including the Memory State Delta Tree Algorithm, which precomputes and store the memory state at different points in the traces and dynamically recreates memory from the last checkpoint up to the desired instruction. Using this algorithm, Atlantis is the first trace analysis environment that can provide efficient memory reconstruction of gigabyte assembly traces in an interactive manner.

Reconstructing functions and processes of a trace create higher-level entities helps analysts cope with the complexity of a trace. This information allows them to observe potential patterns of the program. Atlantis examines the trace to automatically identify both functions and processes and provides specialized views to inspect and navigate them.

II. Tool Description

Atlantis is an interactive environment for analyzing assembly level traces to assists security analysts who perform dynamic analysis of software in search of security vulnerabilities. It is currently capable of inspecting large traces (dozens of gigabytes long). It has four main features[1]:

1) It allows to navigate and inspect each instruction in the trace.
2) It reconstructs the program's memory at any point during the trace (e.g. a byte-level snapshot of memory as seen by the program at that particular moment in time), and provides features to query and navigate the memory state of the program.
3) It provides function and thread views to help analysts cope with the size and complexity of the trace.
4) It provides a powerful search mechanism to query and navigate the trace.

The following subsections describe why we need to pre-process the trace file; The new sub module, Gibraltar, for trace pre-processing; What is new for Atlantis as a trace viewer.

A. The need to pre-process the trace

One of the major challenges of analysing assembly-level traces for security analysis is that they tend to be huge. This makes reconstructing the entire memory state at any point of the trace very costly. Cleary et al [6] described several different strategies to address this problem. They rely upon the creation of data structures that can be pre-computed. Some of them consume more storage while others require more computation at the time when the user wants to inspect memory at a given point. One of these strategies is the Memory State Delta Tree algorithm, which we have implemented in Atlantis. This algorithm strikes a balance between storage and time that makes live interaction possible.

In a nutshell, this algorithm requires the creation of the memory state delta tree which is a B-tree [7]. Each node contains a start instruction line number(startline) and end instruction line number(endline) along with a snapshot delta of the memory change for the section of the trace from startline to endline. The startline of a node is the next number of the endline of its previous node in the same tree level. The parent-node's startline equals to its first child-node's startline while its endline equals to its last child-node's endline.

The memory reconstruction of a certain instruction line is based on that line number search in the B-tree on the endline numbers. All the memory snapshot of the nodes in the traverse path will be retrieved and combined into the memory state of that instruction line. This process reduces the computational complexity of computing the trace at any given point from linear to logarithmic (with respect to the number of instructions in the trace).

However, it has two main drawbacks: computing the memory state delta tree is very expensive (e.g. a gigabyte trace might require hours to be processed) and it requires a lot of disk space to store it. Fortunately, the memory state delta tree needs to be computed only once per trace.

Another challenge that Atlantis users faced was loading a trace for the first time. At this point, the trace was analyzed (i.e. to reconstruct functions and processes) and several indices were created to speed up navigation. As the traces became larger, this process slowed down and became too long to be done interactively.

For these reasons, processing of a trace is now done off-line. Once the trace is generated, it can be processed and all the necessary data structures (including the memory state delta tree and any indices) are created. This process is depicted in Fig. 1. The pre-processing is done by a new module called Gibraltar that reads the original trace and generates a SQLite database contains all that Atlantis requires.

B. Gibraltar: A Trace Pre-Processor

Gibraltar is a pre-processor that converts trace file into a SQLite database that Atlantis uses as input. Gibraltar is responsible for three main tasks: a) creation of the memory state delta tree, b) identification of higher-level entities, such as functions and processes, and c) creation of data structures (such as indices to the instructions in the execution trace) to improve the performance of Atlantis. The output of Gibraltar also contains all the information present in the original trace.

Gibraltar needs to be run only once per trace and does not require any interaction with the user; hence it can be run off-line, right after the trace has been generated.

Separating the pre-process tasks in Gibraltar is an other distinguish improvement that makes the later interactive response from Atlantis to the users possible.

C. Atlantis: A Trace Viewer

Atlantis connects to the SQLite database file generated by Gibraltar. The main views of Atlantis, depicted in Fig.2, presents the state of the program at a particular instruction line. For this instruction line, Atlantis retrieves the tree node data from the database, reconstructs the memory state, and presents it. All the unique and novel features provided by Atlantis are hosted in its views. This views provides various information to the analysts which significantly improve their analysis efficiency. Due to space constraints we only elaborate on the most novel views.

1) Memory views: The reconstructed memory state is updated immediately after the selection of an instruction and can be inspected in the **Register view** and **Memory view**. The memory changes made by the current instruction line are highlighted in red.

2) Search views: Search views provide users with an easier way to access information. Atlantis has two search views. The **Search Assembly view** allows users to search for specific instructions. The **Memory Search view** provides users with two different search expressions, text or hexadecimal. When the users are looking for a specific value, all matching memory locations (with the values they contained at the current line

[1]The last three features are the main contributions of this paper while the first one is presented in the paper in WCRE2012 [5].

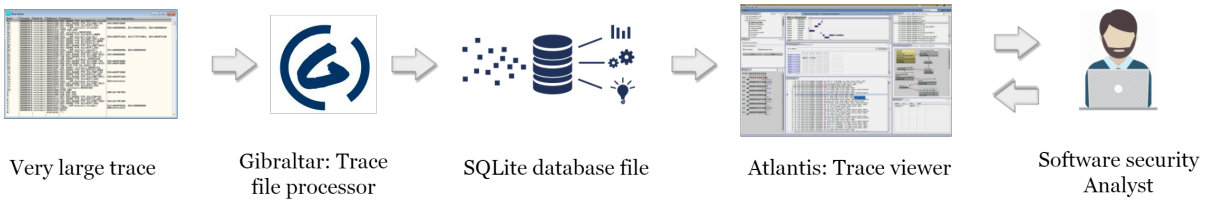

Fig. 1. Before a trace can be inspected by Atlantis, it must be pre-processed offline by Gibraltar. Gibraltar reads the trace and creates a SQLite database that contains the data structures to speed up access to the trace, the memory state delta tree (used for memory reconstruction) and the reconstructed functions and processes.

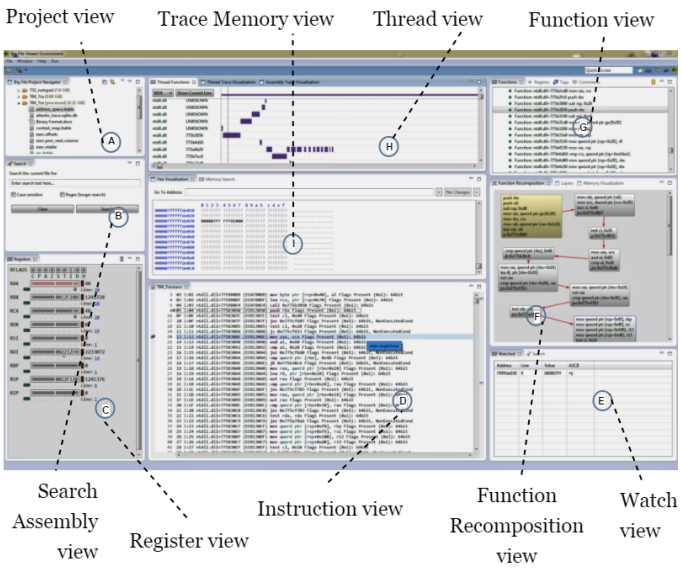

Fig. 2. Screenshot of Atlantis with Default View Setting

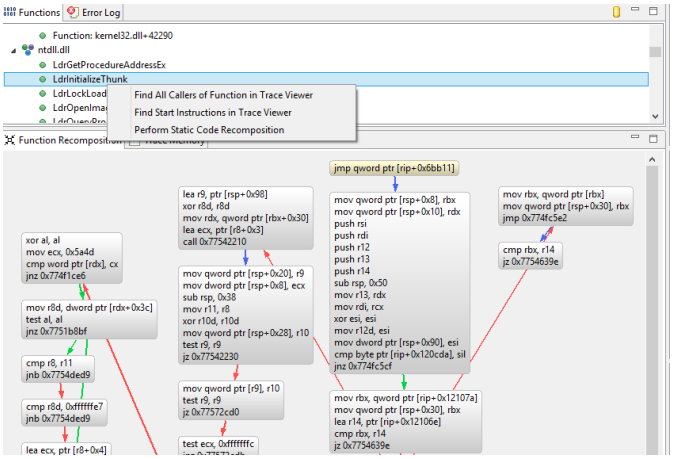

Fig. 3. Screenshot of Function views

in the Instruction view) will be listed in the result window. The Memory List view allows the user to browse specific memory addresses and to identify instructions that modify these addresses.

3) Functions views: There are two views in this group: **Functions view** and **Function Recomposition view**, as shown in Fig. 3. The Functions view lists all the executable modules (such as .dll and .exe) in the trace. Expanding an executable entry shows the functions in this executable that are called in the trace. Atlantis inspects the executable binary or any DLLs it used in search of symbolic names to label the identified functions found in the trace. This higher-level information provides analysts more a structural analysis method.

Users can get the function recomposition information by right clicking a specific function entry and selecting the "Perform Static Code Recomposition" action item as shown in Fig. 3. This is the impressive since it allows analysts to understand the program complexity without source code.

4) Trace Visualization views: There are three views in this group: **Assembly Visualization view**, **Thread Trace Visualization view** and **Thread Functions view**. Fig. 4 shows a screen shot of these three views (the last two are new). The Tread Trace Visualization view shows the threads' temporal relationships. The Assembly Trace Visualization view shows the time segments in which each executable module is being executed. Jumping among executable modules indicates calls across each module. The Thread Functions view shows the function call of the selected thread. Meaningful patterns, might be discovered by experienced reverse engineers using these views.

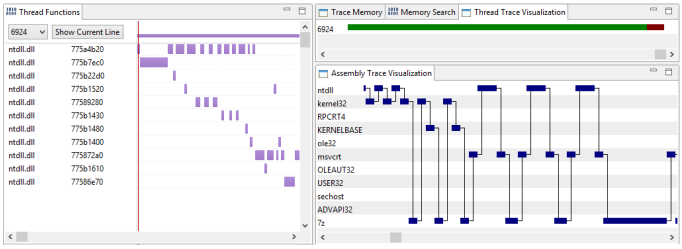

Fig. 4. Screenshot of Visualization views

III. Preliminary Evaluation

Our research partner, Defence Research and Development Canada (DRDC) uses Atlantis. Assembly-level trace files are generated by an in-house tool they have developed. DRDC has provided several traces to measure the performance of Gibraltar and Atlantis. The measurements were conducted separately for Gibraltar and Atlantis, on a machine running

Windows 7, with a 3.60GHz Intel i7-4790 CPU, and 16GB of RAM.

A. Gibraltar

By running the sample traces, we proved that Gibraltar can successfully reconstruct the memory delta tree and other needed data for Atlantis. With regard to precise measurement of Gibraltar, two measures are of interest. First, the time to process a trace. Second, the size of the SQLite database. We conducted the measurement tests on three trace files. Table I shows the results. As one can see, the processing time of Gibraltar is very long. Fortunately it only needs to run once per trace. The size of the output file is much larger than the input, due to the additional information and the indices created. The processing time and output size depend on the characteristics of the specific trace and do not correlate with the trace file size. For example, the total memory footprint of the traced program will affect all three metrics (regardless of actual size of the trace), and thrashing of memory during the execution of the program will also affect processing time.

TABLE I
MEASUREMENTS FOR TRACES WHEN PROCESSED BY GIBRALTAR

Traced Application	Input Trace File Size (GB)	Input Trace Number of Instructions	Processing Time (hours)	Output Database File Size (GB)
AdobeReader	4.81	82,778,317	12.44	29.5
Cmd	23.3	2,772,154	63	150
Chrome	38.8	671,168,459	212	245

B. Atlantis

As we claim before, Atlantis can deal with large traces responsively. Since the biggest bottle neck of the response time of Atlantis is located in the reconstruction of memory state, we only evaluate its responsiveness by measuring the time it takes to update its memory view (the most time-consuming view, all over views are updated almost instantaneously) when a user jumps to a given instruction in a very large trace. This measurement will be an indicator of how interactive Atlantis is. For this test, we used the trace file of Cmd (one of the three shown in Table I). We instrumented Atlantis to measure the time it takes from selecting an instruction to the time the memory view has been completely updated. To perform a measurement, we placed the current and destination instruction pairs in the Instruction view. We navigated from the current to the destination using the "Go To Line" short-cut provided by the Instruction view. This minimized any interaction with other Atlantis features that might skew the results.

We divided the test into three groups, each with different distances between the source and destination instructions. Each group has a constant gap (in terms of number of instructions) between the current and destination instructions; these are 13,861, 138,607 and 831,646, corresponding to 0.5%, 5% and 30% of the length of the trace file.

We performed the test in both directions: forward and backward. Forward means the current instruction number is less than the destination instruction number, while backward is the opposite. We performed 10 tests in each group per direction and recorded the time it took to reconstruct the memory and update the memory view for each test. Table II shows the minimum, maximum and average time of each test group and direction. Miller [8] described that a response time of 100 ms is perceived as instantaneous while 1 second or less is fast enough for users to feel they are interacting freely with the computer. From the result of our measurement test, we can see even the maximum response times are far less than 1 second. These results support the view that Atlantis provides a responsive user experience.

TABLE II
ATLANTIS INTERACTION TIMES, IN SECONDS, FOR MEMORY RECONSTRUCTION.

line gap	13,861 (0.5%)		138,607 (5%)		831,646 (30%)	
Direction	FW[1]	BW [2]	FW	BW	FW	BW
Max(s)	0.159	0.204	0.217	0.272	0.151	0.168
Min(s)	0.031	0.094	0.056	0.081	0.132	0.063
Avg(s)	0.098	0.14	0.14	0.152	0.139	0.119

[1] Forward. [2] Backward.

Due to the space limitation, we can not present evaluation of other futures. The effect of these futures can be refer from the demo video.

IV. CONCLUSION AND FUTURE WORK

This paper presents the significantly improved Atlantis. Its new given ability of deriving and visualizing various types of information, especially the memory state at any point of the trace, the function reconstruction and the call flow helps analysts understand the program's behaviour and reduce the cognitive overload of dealing with these very large traces.

We have also shown that by pre-processing the trace (a step that can be performed right after the trace is created, without any interaction from the user), Atlantis is capable of providing real-time views of the memory state of the program at any point of a huge trace.

In the future, we aim to extend our tool in two directions. First, we would like our pre-processor to read traces from other trace generators. This will permit users of other trace generators to use Atlantis. And this is the most important step and reason for Atlantis open sourcing while it is now only an in-house tool. Second, we want to assist in the analysis and visualization of dual-traces. A dual-trace consist of two execution traces that are generated from two applications that are communicating in some way. Applications nowadays rarely work in isolation, and many are designed as collaborative systems or modules in a network [9], which makes the discovery of vulnerabilities even harder since communications and interactions across applications affect their behaviour.

ACKNOWLEDGMENT

We wish to thank all members in CHISEL lab especially Cassandra Petrachenko, Alexey Zagalsky, Omar Elazhary and our visiting professor Andy Zaidman for supporting the development of this tool and the writing of this paper.

REFERENCES

[1] V. M. Igure and R. D. Williams, "Taxonomies of attacks and vulnerabilities in computer systems," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 1, 2008.

[2] C. P. Pfleeger and S. L. Pfleeger, *Security in computing*. Prentice Hall Professional Technical Reference, 2002.

[3] M. Dowd, J. McDonald, and J. Schuh, *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[4] T. Ball, "The concept of dynamic analysis," in *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6. Springer-Verlag, 1999, pp. 216–234.

[5] B. Cleary, M.-A. Storey, L. Chan, M. Salois, and F. Painchaud, "Atlantis-assembly trace analysis environment," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 505–506.

[6] B. Cleary, P. Gorman, E. Verbeek, M.-A. Storey, M. Salois, and F. Painchaud, "Reconstructing program memory state from multi-gigabyte instruction traces to support interactive analysis," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 42–51.

[7] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[8] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 1968, pp. 267–277.

[9] L. Wen, D. Kirk, and R. G. Dromey, "Software systems as complex networks," in *Cognitive Informatics, 6th IEEE International Conference on*. IEEE, 2007, pp. 106–115.