Defence Research and Development Canada | Recherche et développement pour la défense Canada

CAN UNCLASSIFIED

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

Huihui Huang
Eric Verbeek, Daniel German, Margaret-Anne Storey,
University of Victoria

Martin Salois
DRDC – Valcartier Research Centre

Bibliographic information: 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17-24 September 2017

Date of Publication from Ext Publisher: September 2017

Defence Research and Development Canada

External Literature (P)
DRDC-RDDC-2018-P015
February 2018

Canada

IMPORTANT INFORMATIVE STATEMENTS

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

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

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

***Abstract*—Assembly execution trace analysis is an effective approach for discovering potential software vulnerabilities. However, the size of the execution traces and the lack of source code makes this a manual, labor-intensive process. 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 a program to assist in the identification of patterns of interest. The tool we present in this paper, Atlantis, is our trace analysis environment for multi-gigabyte assembly traces, and it contains a number of new features that make it particularly successful in meeting this goal. The contributions of this continuous work fall into three main categories: a) the ability to efficiently reconstruct and navigate the memory state of a program at any point in a trace; b) the ability to reconstruct and navigate functions and processes; and c) a powerful search facility to query and navigate traces. These contributions are not only novel for Atlantis but also for the field of assembly trace analysis. Software is becoming increasingly complex and many applications are designed as collaborative systems or modules interacting with each other, which makes the discovery of vulnerabilities extremely difficult. With the novel features we describe in this paper, our tool extends the security analyst's ability to investigate vulnerabilities of real-world large execution traces and can lay the groundwork for supporting trace analysis of interacting programs in the future.**

Screencast link: https://youtu.be/1s4gNnFf-o4

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 the creation of 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 where one attempts to find vulnerabilities in 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 analyzes the execution a running program, is one of the main methods used in vulnerability detection. A subset of dynamic analysis is called assembly tracing, which records all the executed processor instructions, often with additional information such as memory and disk accesses. The traces we are examining in this paper 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, analyzing these traces is a labor-intensive task.

The typical usage scenario in such semi-automated vulnerability research goes like this [5]: fuzz the program, finding thousands of crashes; rerun those crashes through heuristics to find the most promising ones; trace the promising ones; run various automated analysis on those traces (e.g. taint analysis [6]) to figure out if the crash can lead to an exploit; analyze the remaining traces manually for confirmation of exploitability. Obviously, we want those final traces to be as small as possible, to reduce the load both on computing and human resources—and the execution time of the program is usually very short, just long enough to load a file or parse some communications—but, chances are, there will still be very large traces to look at. Enter Atlantis.

Atlantis is an assembly trace analysis environment. It was initially designed to provide security engineers with the ability to inspect and navigate large traces. Users are able to browse a trace, add comments, and mark regions of interest: a first version of Atlantis was demonstrated at WCRE 2012 [7].

This paper showcases our more recent contributions to assembly execution trace analysis implementing several new powerful and novel features in Atlantis. As far as we know, these newly-implemented features make Atlantis the only tool that can reconstruct the memory state of a trace as well as visualize function call patterns and process and executable interaction patterns. All of these unique and novel features allow software security analysts to gain new insights in 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 the state of the memory used by the program at any point in the trace; what memory has the program accessed, and what are its current contents. This feature allows analysts to observe how a program accesses and changes its memory to discover potential vulnerabilities.

However, implementing this feature is not trivial. Cleary et al. [8] proposed several methods to solve this problem,

This work is funded by a Canadian DND/NSERC grant

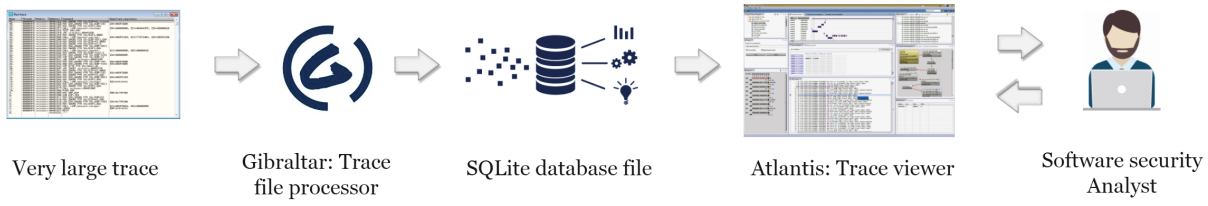

Fig. 1. Before Atlantis can inspsect a trace, it must be preprocessed by Gibraltar offline. Gibraltar reads the trace and creates an SQLite database that contains the data structures to speed up access to the trace, the memory state delta tree (used for memory reconstruction), and the reconstructed functions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

including the Memory State Delta Tree Algorithm, which precomputes and stores the memory state at different points in the trace 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 creates higher-level entities that help analysts cope with the complexity of the trace. This information allows them to observe potential patterns in a program. Atlantis examines a trace to automatically identify both functions and processes and provides specialized views for inspection and navigation.

II. Tool Description

Atlantis is an interactive environment for analyzing assembly level traces. It can assist security analysts performing dynamic analysis of software in search of security vulnerabilities. Atlantis can currently inspect traces that are dozens of gigabytes long, limited only by computer memory. It has four main features, the last three features of which are the main contributions of this paper, while the first one was presented in our WCRE2012 [7] paper:

1) It allows analysts to navigate and inspect each instruction in a trace.
2) It reconstructs a program's memory at any point in a trace (i.e., a byte-level snapshot of memory 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 a trace.
4) It provides a powerful search mechanism for querying and navigating a trace.

The following subsections describe Atlantis in further detail and discuss certain technical requirements.

A. Preprocessing Traces

One of the major challenges of analyzing traces at the assembly level 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 [8] described several strategies to address the memory state reconstruction problem. Some of these strategies consume more storage for the created data 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, it creates a memory state delta tree, which is a B-tree [9]. 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 lien number of the endline of its previous node at the same tree level. The parent node's startline is equal to its first child node's startline, while its endline is equal 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. 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, this algorithm has two main drawbacks: computing the memory state delta tree is very expensive (a gigabyte trace might require hours of processing) 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, so it can be preprocessed in the fuzzing chain.

Another challenge for trace analysis tool is that, when user navigating around the trace, the tool has to be able to respond with the analysis information immediately. As the traces became larger, this process slowed down and became too long to be done interactively. It was decided to also preprocess this to reconstruct functions and processes, and to create several indices to speed up navigation.

All of this preprocessing is performed by a new module called Gibraltar that reads the original trace and generates an SQLite database that contains all the information Atlantis requires (Fig. 1). Gibraltar converts a trace file into an SQLite database that is fed into Atlantis. 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) to improve the performance of Atlantis. The database contains all the information from the original trace.

Gibraltar needs to be run only once per trace and does not require any interaction from the user. Ideally, it is run offline as part of the fuzzing chain, right after a trace has been generated.

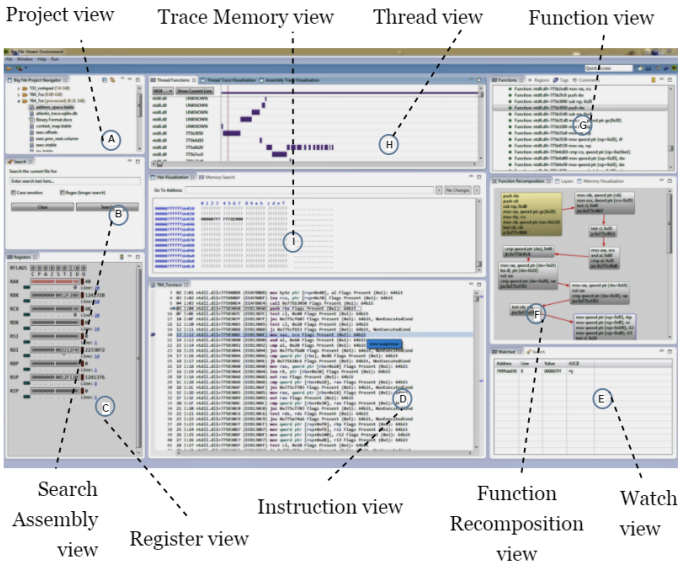

Fig. 2. Screenshot of Atlantis with default view

B. Atlantis: Trace Viewer

The Atlantis trace viewer connects to the SQLite database file generated by Gibraltar. The main views of Atlantis (Fig. 2) present the state of the program at a particular instruction line. For this instruction line, Atlantis retrieves the tree node data from the database and reconstructs the memory state. Atlantis' unique and novel views provide various types of information to the analyst. Due to space constraints, we only elaborate on the most novel views.

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

2) Search views: Three search views provide an easier way to access information. The *Search Assembly* view allows users to search for specific instructions. The *Memory Search* view provides different search expressions, text or hexadecimal, to search memory changes in the memory state of the current instruction line. When a user searches for a specific value, all matching results will be listed in the result window and are navigable to the Memory view. The *Memory List* view allows the user to browse specific memory addresses and to identify instructions that modify these addresses.

3) Function views: There are two views in this group, *Function* and *Function Recomposition* (Fig. 3). The *Function* 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 uses in search of symbolic names to label the identified functions found in the trace. This higher-level information provides a more structured method for trace analysis.

Users can get the function recomposition information by right clicking a specific function entry and selecting "Perform Static Code Recomposition" (Fig. 3).

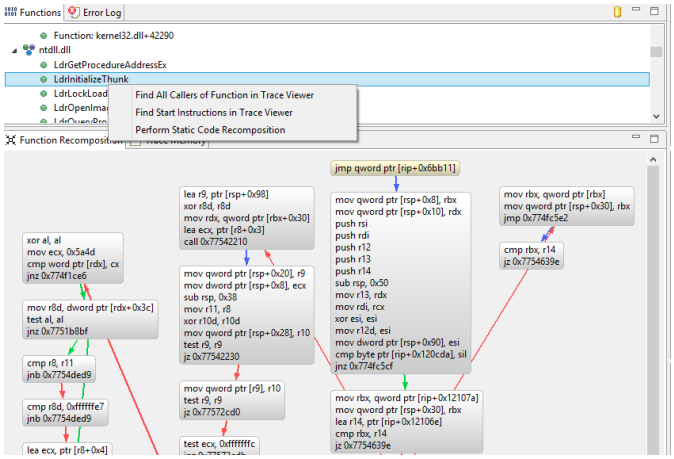

Fig. 3. Screenshot of Function view

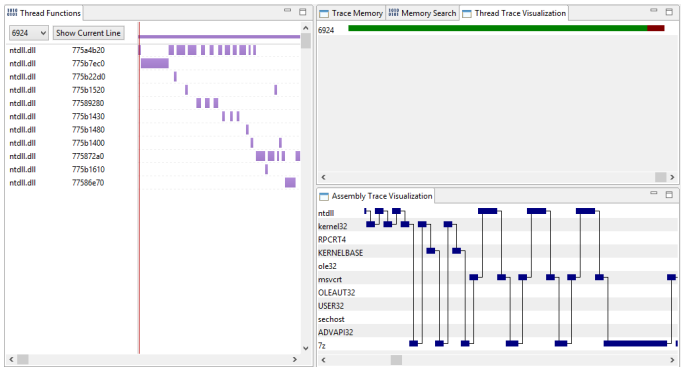

Fig. 4. Screenshot of Visualization views

4) Trace Visualization views: There are three views in this group, *Assembly Visualization*, *Thread Trace Visualization* and *Thread Functions*. Fig. 4 shows a screenshot of these three views—the last two are new. The *Thread Trace Visualization* view shows the temporal relationships of the threads. 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 calls of the selected thread. Meaningful patterns can be discovered by experienced reverse engineers using these views.

III. Preliminary Evaluation

Our research partners at Defence Research and Development Canada (DRDC) have been testing Atlantis. They generated several assembly level trace files with an in-house tracer and provided them to measure the performance of Gibraltar and Atlantis. The measurements were conducted on a machine running Windows 7, with a 3.60GHz Intel i7-4790 CPU and 16GB of RAM.

A. Gibraltar

Running the sample traces, we demonstrated that Gibraltar can successfully reconstruct the memory delta tree and other

TABLE I
MEASUREMENTS FOR TRACES 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

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.

needed Atlantis data from large trace files. With regard to precise measurement of Gibraltar, two measures are of interest. First, the time to process a trace, and second, the size of the SQLite database. We conducted the measurement tests on three trace files. Table I shows the results. As mentioned, Gibraltar takes a long time to process a trace. Fortunately, it only needs to run once per trace and it can be preprocessed as part of the fuzzing chain. It is theoretically possible to make the traces much smaller by only tracing interesting points of the program instead of tracing everything, but this is not the job of Atlantis. Atlantis must scale to very large traces in case they are needed and this is the research challenge we address.

The size of the output file is much larger than the input due to the additional information and indices created. The processing time and output size depend on the characteristics of the specific trace and do not directly correlate with the trace file size. For example, the total memory footprint of the traced program will affect all three metrics (regardless of the actual size of the trace), and the amount of memory the traced program allocates and releases will greatly affect size and processing time.

B. Atlantis

As we claimed before, Atlantis can deal with large traces in a responsive manner. Since Atlantis' biggest response time bottleneck is located in the reconstruction of memory state, we only evaluate Atlantis responsiveness by measuring the time it takes to update its *Memory* view when a user jumps to a given instruction in a very large trace. This is by far the most time-consuming view, as all other views are updated almost instantaneously. 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 instructions using the "Go To Line" shortcut. 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 respectively.

We performed the test going forward and backward: forward means the current instruction number is less than the destination instruction number; 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 in each direction. Miller [10] 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 freely interacting with the computer. From the results of our test, we can see that the maximum response times are far less than 1 second. These results support the view that Atlantis provides a responsive user experience.

Due to space limitations, we cannot elaborate on other features of Atlantis. For further information, please see the demonstration videos that complement this paper[1].

IV. CONCLUSION AND FUTURE WORK

This paper presents the improvements we have made to Atlantis to allow one to derive and visualize various types of information. Knowing the memory state at any point in a trace, the function reconstruction, and the call flow help analysts understand a program's behaviour and reduce the cognitive overload of dealing with these very large traces.

We have also shown that by preprocessing traces—a step that can be performed right after the trace is created, as part of the fuzzing chain and without any interaction from the user—Atlantis is capable of providing real-time views of the memory state of the program at any point in a huge trace.

In the future, we aim to extend our tool in two directions. First, we would like our preprocessor to read traces from other trace generators (and to be ever faster, of course!) This will allow users of other trace generators to use Atlantis, which we are in the process of releasing under an open source license. Second, we want to assist in the analysis and visualization of dual traces, defined as traces generated from two applications that are communicating in some way. Applications (and malware!) nowadays rarely work in isolation, and many are designed as collaborative systems or modules in a network [11], which makes the discovery of vulnerabilities even harder as these communications greatly affect their behaviour.

[1] https://youtu.be/wTTENxgwEvc

ACKNOWLEDGMENT

We wish to thank Cassandra Petrachenko, Alexey Zagalsky, Omar Elazhary, and Andy Zaidman for their help in the preparation 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] Cybersecurity Ventures. (2017, Jan.) Zero Day Report: A Special Report from the Editors at Cybersecurity Ventures. [Online]. Available: http://cybersecurityventures.com/zero-day-vulnerabilities-attacks-exploits-report-2017

[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] P. Godefroid, M. Y. Levin, and M. David, "SAGE: Whitebox Fuzzing for Security Testing," *acmqueue*, vol. 10, no. 1, Jan. 2012.

[6] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," Internet Society, Feb. 2005. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/05/proceedings

[7] 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.

[8] 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.

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

[10] 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.

[11] Trend Micro. (2016, Dec.) Security predictions: The next tier. [Online]. Available: https://www.trendmicro.com/vinfo/us/security/research-and-analysis/predictions/2017

DOCUMENT CONTROL DATA

(Security markings for the title, abstract and indexing annotation must be entered when the document is Classified or Designated)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g., Centre sponsoring a contractor's report, or tasking agency, are entered in Section 8.)

DRDC – Valcartier Research Centre
Defence Research and Development Canada
2459 route de la Bravoure
Quebec (Quebec) G3J 1X5
Canada

2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.)

CAN UNCLASSIFIED

2b. CONTROLLED GOODS

NON-CONTROLLED GOODS
DMC A

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

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

4. AUTHORS (last name, followed by initials – ranks, titles, etc., not to be used)

Huihui, H.; Verbeek, E.; German, D.; Storey, M.-A.; Salois, M.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
September 2017	5	11

7. DESCRIPTIVE NOTES (The category of the document, e.g., technical report, technical note or memorandum. If appropriate, enter the type of report, e.g., interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

External Literature (P)

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

DRDC – Valcartier Research Centre
Defence Research and Development Canada
2459 route de la Bravoure
Quebec (Quebec) G3J 1X5
Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
05AA	SRE14-046

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC-RDDC-2018-P015	

11a. FUTURE DISTRIBUTION (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Public release

11b. FUTURE DISTRIBUTION OUTSIDE CANADA (Any limitations on further dissemination of the document, other than those imposed by security classification.)

12. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

Assembly execution trace analysis is an effective approach for discovering potential software vulnerabilities. However, the size of the execution traces and the lack of source code makes this a manual, labor-intensive process. 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 a program to assist in the identification of patterns of interest. The tool we present in this paper, Atlantis, is our trace analysis environment for multi-gigabyte assembly traces, and it contains a number of new features that make it particularly successful in meeting this goal. The contributions of this continuous work fall into three main categories: a) the ability to efficiently reconstruct and navigate the memory state of a program at any point in a trace; b) the ability to reconstruct and navigate functions and processes; and c) a powerful search facility to query and navigate traces. These contributions are not only novel for Atlantis but also for the field of assembly trace analysis. Software is becoming increasingly complex and many applications are designed as collaborative systems or modules interacting with each other, which makes the discovery of vulnerabilities extremely difficult. With the novel features we describe in this paper, our tool extends the security analyst's ability to investigate vulnerabilities of real-world large execution traces and can lay the groundwork for supporting trace analysis of interacting programs in the future.

---

n/a

13. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g., Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

big data, assembly, visualisation