Reconstructing Program Memory State from Multi-gigabyte Instruction Traces to Support Interactive Analysis

Brendan Cleary, Patrick Gorman,
Eric Verbeek, Margaret-Anne Storey
University of Victoria
Victoria, BC, Canada
bcleary@uvic.ca

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

***Abstract*—Exploitability analysis is the process of attempting to determine if a vulnerability in a program is exploitable. Fuzzing is a popular method of finding such vulnerabilities, in which a program is subjected to millions of generated program inputs until it crashes. Each program crash indicates a potential vulnerability that needs to be prioritized according to its potential for exploitation. The highest priority vulnerabilities need to be investigated by a security analyst by re-executing the program with the input that caused the crash while recording a trace of all executed assembly instructions and then performing analysis on the resulting trace. Recreating the entire memory state of the program at the time of the crash, or at any other point in the trace, is very important for helping the analyst build an understanding of the conditions that led to the crash. Unfortunately, tracing even a small program can create multi-million line trace files from which reconstructing memory state is a computationally intensive process and virtually impossible to do manually. In this paper we present an analysis of the problem of memory state reconstruction from very large execution traces. We report on a novel approach for reconstructing the entire memory state of a program from an execution trace that allows near real-time queries on the state of memory at any point in a program's execution trace. Finally we benchmark our approach showing storage and performance results in line with our theoretical calculations and demonstrate memory state query response times of less than 200ms for trace files up to 60 million lines.**

I. Introduction

An exploit occurs when a vulnerability in a software program is discovered and malware exploiting the flaw is developed. Software security analysts attempt to protect against exploits by proactively finding vulnerabilities in programs before malware authors find and develop code that exploits those vulnerabilities.

The process of determining if a given program is susceptible to exploitation is called exploitability analysis. Security analysts typically use a combination of static and dynamic analysis techniques to determine if a program is vulnerable to exploitation. One common method is fuzzing [21], which attempts to deliberately cause the program to crash by subjecting the program to millions of mutated or generated inputs. Crashes are then prioritized using *ad hoc* techniques [15] with the highest priority crashes then analyzed for exploitability. One analysis technique is to re-execute the program with the input that caused the crash, record a trace of all executed assembly instructions up to the crash and then perform analysis on the resulting trace. The resulting execution trace is analyzed to find the root cause of the crash and to assess if an exploit could take control of the program's execution. This last step requires a great deal of human reasoning.

Currently, security analysts rely on off-the-shelf text editors and comparison tools when performing manual analysis of these traces. In previous work [5], we developed the Atlantis assembly trace analysis environment designed specifically to work with execution traces generated during exploitability analysis. Atlantis provides security engineers with a user-friendly environment to perform, record and share their analyses. An important aspect in these analyses and in determining a program's potential explotability from a trace is being able to quickly rewind the program's execution to reconstruct the state of the memory at different points during the execution. This allows the analyst to observe how the memory state of a program evolves in response to the instructions executed.

Unfortunately, even for programs of modest size, the traces generated when performing exploitability analysis can be hundreds of millions of lines in length with each instruction possibly causing a change in the memory state of the program. Reconstructing the memory state of a program at a particular point in a trace is a computationally intensive process, requiring simulation of all memory operations from the start of the program's execution to the desired point in the trace. Aside from trivially small example programs, doing this calculation in real-time can take many minutes of processing. This latency in reconstructing memory state prevents the types of interactive real time analysis required by security analysts. Removing this latency has the potential to dramatically enhance the work practices of security analysts.

In this paper we present a method by which we analyze all memory and register references captured in a program's execution trace to allow us to pre-compute an indexed memory state for each trace instruction. Pre-computation of the memory index involves building a structure that we call a memory state

978-1-4799-2931-3/13/$31.00 © 2013 IEEE

WCRE 2013, Koblenz, Germany

delta tree. This data structure then enables us to serve requests from a user for the entire memory state of the program at arbitrary points in the trace in almost real time. This allows analysts to interactively browse the historical memory state of a program. Finally, we present benchmarks of the query response performance of our technique where we demonstrate response times of less than 200ms for memory state queries at arbitrary locations in traces of up to 60 million lines.

II. Problem Definition

Atlantis is an integrated assembly trace analysis environment designed to assist security engineers perform and manage trace analysis [5]. Atlantis was developed in collaboration with security analysts to meet their requirements and support their work practices [22]. To this end, Atlantis supplements many of the features of modern IDEs with novel annotation and navigation techniques to support trace-based exploitability analysis. One of the key features of Atlantis is its ability to allow users to manipulate very large trace files, in the order of tens of millions of lines. Another key feature are the Atlantis memory and register views which allow a user to investigate the state of the traced program's memory and registers as they navigate through these very large traces in real time. This paper describes the theoretical underpinnings of how these views are constructed.

Our use case for reconstructing memory and register state can be summarized as follows. Given a tool such as Atlantis, when the user selects a particular trace line we want to present that user with tables of all memory addresses and registers referenced by the program up to that point in the trace with their current values. The system must return the required information quickly enough to allow the user to interactively navigate through the trace, observing the memory and register state updating as they navigate. Users should be able to jump to any point in the trace or "step" through the trace line by line. To provide an optimal user experience, results should be displayed in less than 500ms. We chose this response time target in consultation with our research client the DRDC and because it was a compromise between the 100ms response time limit where users perceive a system as responding instantaneously, and the 1 second response time limit above which a user's flow of thought is interrupted [16].

More formally, given a program P and a trace T consisting of an ordered set of n executed instruction records (trace lines) $T = \{i_1, i_2, i_3 ... i_n\}$ generated by the tracing of P, we want to compute M, the state of the memory of P at instruction record i_x in T. Where M is a set of memory address and value pairs: $M = \{\{a_1, v_1\}, \{a_2, v_2\}, \{a_3, v_3\} ... \{a_m, v_m\}\}$. To calculate M we require that the order of the sequence of trace instruction records in T reflects the exact order of execution regardless of which thread they were executed in (that is, we require that the trace be totally sequential, without concurrent constructs), and that each instruction record details all memory addresses and registers modified by that instruction and the values assigned.

As an example consider the sample trace file in Table I. Here we see six instruction records where each contains a unique sequential instruction record id, the instruction disassembly, and the sets of registers and memory addresses modified by that instruction along with the values assigned to those addresses by the instruction.

After a user selects a line of the trace, we want to generate two tables, one containing the memory addresses referenced by the program with their current values, and a second containing the registers and their values. Tables II and III depict how the contents of these memory and register tables would evolve if the user were to navigate line by line through the example trace in Table I. Constructing these tables for a small example is not complex. Reading the trace from start to finish, as we encounter references to memory addresses and registers and if we have not already seen a reference to that address or register, we add it to the table. If we have already seen that address or register, we update its value. For example, line 5 of the example trace changes the value of an already referenced memory address. In this case we simply update the value in the memory state table rather than adding a new row.

While constructing the memory and register reference tables in this manner for small examples is trivial, it is not possible to construct these tables for large multi-gigabyte program traces so that users can step through the trace while observing memory and register state updates in real time (that is, within the response time threshold of 500ms required by our use case).

III. Memory State Reconstruction Strategies

There are many possible approaches to reconstructing a program's entire memory state at a given location in a trace. In this section we summarize some of these strategies, illustrating how each would work and then discussing the advantages and drawbacks of each approach in the context of very large traces.

A. Linear Reconstruction

Linear Reconstruction (LR) is the most naive method of reconstructing memory state and operates similarly to the example method described in section II. The LR strategy computes the memory state at instruction record i_x through a simple linear traversal of the trace from instruction i_0 to instruction i_x. Storing references to memory addresses or registers in an in-memory data structure that represents the memory state of the program under study. This simple strategy can compute the memory state of the program in $O(x)$. While this translates to very fast response times for very small traces entirely held in memory, as x increases to tens of millions of instructions that need to be paged from disk and parsed, the response time quickly becomes too great for real-time interactivity.

Another problem with the LR approach is the lack of a caching strategy. Using this strategy, each time the user selects a new target trace instruction record, the memory state has to be recomputed. When the new target instruction record is later in the trace than the previous target instruction record, the

TABLE I
EXAMPLE TRACE FILE EXCERPT

Id	Disassembly	Registers	Memory
1	(672ca571) mov ebp, esp	EBP=0012f9d0	
2	(672ca573) sub esp, 0x8	ESP=0012f9c8	
3	(672ca576) cmp dword ptr [0x672cb524], 0x0		[672cb524]=00000000
4	(672ca57d) jnz 0x672ca58e		
5	(672ca57f) mov dword ptr [0x672cb524], 0x1		[672cb524]=00000001
6	(672ca589) call 0x672ca520	ESP=0012f9c4	[0012f9c4]=672ca58e

existing computed state can be reused with just the instruction records between the old target and new target needing to be traversed. However, when the target instruction record is earlier in the trace than the previous target, the memory state has to be recomputed by un-applying memory references from the now irrelevant trace instruction records. In the worst case, both these situations will require that the memory state be completely recomputed from i_0 to i_x. Further, due to the simplistic memory model, previously computed state is not stored. For example, the memory state calculated for the entire trace from i_0 to i_n will be overwritten by the memory state for i_0 to i_1, requiring subsequent queries for i_0 to i_n to be recomputed entirely.

A third issue with the LR strategy is the amount of redundant computations that are not of interest to the user. For a given location in the trace, the corresponding memory state can only have a single value for each memory address and each register (the last value stored in that location prior to and including i_x). Yet under the naive LR strategy, each memory address and register value has to be calculated for all trace instructions from i_0 to i_x. This means that a lot of the work carried out may ultimately end up being redundant to the memory state at i_x as memory addresses are reused and updated. Reverse traversal of the trace, from i_x to i_0, does not entirely solve this issue either, as in the worst case it still requires a complete traversal.

This naive LR strategy can be improved using a more complex memory state model that allows for caching of previously calculated state. However, depending on the memory footprint of the program under study, this may quickly exceed the available system memory and would need to be paged to disk, at which point the LR strategy evolves into a class of more complex strategies.

B. Cached Snapshots

The Cached Snapshots (CS) strategy uses a single linear traversal of the trace from beginning to end to calculate the entire memory state of the program under study at each line and then stores that set of memory states to disk as individual snapshots. In the worst case, this strategy results in n cached snapshots of the memory state, where n is the number of trace instruction records (trace lines). Restoring the memory state for a given trace instruction record i_x is then a simple reading of the corresponding snapshot from disk into memory and presenting it to the user. This method also has the advantage that portions of a memory state snapshot can be paged into main memory on demand, which means that the program under study can have a larger memory footprint than the analyst's workstation.

The drawbacks of this approach are the potentially large amounts of disk space required to store the memory state snapshots and the large amount of disk IO in the preprocessing step. Estimating the total disk space required (without any compression or diff-ing of snapshots) requires knowing the maximum memory footprint of the program under study and the point in the trace at which all relevant memory addresses have been referenced.

For example, if the first y instruction records in a trace each reference a single unique memory address and if each subsequent instruction in the trace operates on one of those addresses, then the number of address-value pairs required to represent the memory state for a trace of n instruction records can be calculated by the following formula:

$$\frac{(y+1)y}{2} + ((n-y)y)$$

The first part in this formula represents the minimum number of address-value pairs required to represent memory states of a program initializing a set amount of memory using y instructions. The second part is the worst case number of address-value pairs that we need to represent the memory states of the rest of the program if every instruction from y to n modifies some memory address. For example, given a program with a memory footprint of 1MB and a 60M line trace, and assuming that 131,072 assembly instructions (1MB / 8-bytes per instruction) are required to initialize the 1MB of memory used by the program, and that each subsequent instruction modifies some memory address, then the total number of memory address-value pairs that need to be stored are approximately 7.8 x 10^{12} or 7.8 trillion. To calculate the amount of space required to store this information on disk, if we assume each address-value pair requires 8-bytes for address plus 8-bytes for value, then the total amount of space required would be 7.8 trillion times 16 bytes which equals approximately 114TB. Estimating disk usage for real world examples is made more difficult by the fact that not all instructions will make references to memory addresses and the instruction in the trace at which all referenced addresses have been observed will be variable. Estimated disk IO times in the preprocessing step can be calculated similarly. For example, if a program has a memory footprint of 1MB with approximately 60M snapshots referencing the full 1MB memory footprint

TABLE II
EXAMPLE MEMORY STATE BY TRACE LINE

Line 1		Line 2		Line 3	
Address	Value	Address	Value	Address	Value
				672cb524	00000000
Line 4		**Line 5**		**Line 6**	
Address	Value	Address	Value	Address	Value
672cb524	00000000	672cb524	00000001	672cb524	00000001
				0012f9c4	672ca58e

TABLE III
EXAMPLE REGISTER STATE BY TRACE LINE

Line 1		Line 2		Line 3	
Address	Value	Address	Value	Address	Value
EBP	0012f9d0	EBP	0012f9d0	EBP	0012f9d0
		ESP	0012f9c8	ESP	0012f9c8
Line 4		**Line 5**		**Line 6**	
Address	Value	Address	Value	Address	Value
EBP	0012f9d0	EBP	0012f9d0	EBP	0012f9d0
ESP	0012f9c8	ESP	0012f9c8	ESP	0012f9c4

and given a disk write speed of 481MB/s (2012 Solid State Disk benchmark [9]), we could write 481 snapshots to disk per second. To sequentially write 55M snapshots would take 124,740 seconds or approximately 1.4 days!

The CS strategy as described is, like the LR strategy, quite naive and not practical. An area for improvement is removing the redundancy between subsequent memory state snapshots by storing just the differences or deltas of previous snapshots rather than the whole snapshot. This would dramatically reduce both disk storage and IO but at the cost of increasing preprocessing complexity and reducing read performance by having to reconstruct memory state from deltas. The next section looks at a smarter snapshotting strategy that attempts to balance preprocessing costs, disk space and response times.

C. Block Snapshot Deltas

The LR and CS strategies mark two extremes on a memory state reconstruction computation - IO spectrum. The LR strategy being computationally intensive but not IO bound, and the CS strategy being storage and IO intensive but capable of very fast response times. In this section we discuss a middle ground strategy that attempts to combine both approaches in an effort to satisfy our use case of keeping response times in the sub-500ms category while balancing storage and IO costs.

One of the ways in which the LR strategy could be best improved is to introduce some kind of memory state caching to reduce the amount of irrelevant computation and prevent the need for memory state to be calculated more than once. However, as we see with the CS strategy, too much caching can result in very high storage and IO requirements. Likewise, one of the ways to improve the CS strategy is to store snapshot deltas rather than full snapshots, again attempting to avoid storing too many deltas which would require significant computation to reconstruct. This strategy, which we call Block Snapshot Deltas (BSD), achieves these goals by combining snapshotting, snapshot deltas and a limited amount of linear calculation.

Conceptually this approach divides the trace into fixed size blocks. Then we proceed to calculate the memory state for the trace line by line. When we reach the end of each trace block, we create and save a memory snapshot delta for that block to disk as a set of textual address-value pairs. Reconstructing the memory state for a given instruction record i_x is then a process of retrieving all snapshot deltas for blocks preceding i_x and then calculating the memory state between the end of

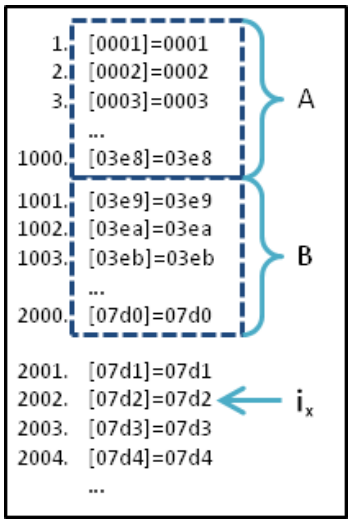

Fig. 1. Calculating Memory State Using Block Snapshots Deltas

the last block and i_x. Figure 1 illustrates this approach for an example trace. Here we see a trace with two blocks defined, A and B. Calculating the memory state at instruction i_x requires retrieving the snapshots for blocks A and B and combining the deltas into a memory state for the end of block B. Using this memory state for the end of block B as a basis we can calculate the memory state at i_x by a simple linear traversal from the end of block B to i_x.

This strategy has several advantages over both LF and CS. First, in comparison to the LF strategy, by making use of cached snapshots, instead of a complete traversal from i_0 to i_x we limit the number of trace instruction records for which we have to recalculate memory state to x minus the end of the previous block y. Second, if we change i_x, we don't have to discard our entire memory state. For example, if i_x was to move earlier in the trace into block B, instead of recalculating from i_0 we just calculate memory state from the end of block A to the new value of i_x in B. Likewise if i_x moves to later in the trace, we just read in the additional required blocks that have not already been read into memory. This strategy effectively limits the number of trace instruction records whose memory state has to be re-computed to $<= blocksize$, where $blocksize$ is the number of trace instruction records which we choose to define as a block. As such, $blocksize$ is an adjustable parameter of the BSD strategy and can be used to balance

processing and IO costs. For example, setting $blocksize$ to a lower value will result in a larger number of smaller snapshots, while increasing it will create a smaller number of larger snapshots, resulting in a knock-on effect on processing, storage and IO costs.

In comparison to the CS strategy, our approach dramatically reduces the amount of IO and disk space used to cache memory state snapshots. This is accomplished in two ways. First, by increasing the granularity of when we choose to make snapshots we can control the number of snapshots that have to be written to disk. Second, by storing snapshot deltas rather than complete snapshots, we reduce both the number of IOs and the amount of disk space required to store those snapshots. While this strategy produces significant savings both in terms of preprocessing time and disk usage, it will of course result in slower response times compared to the pure CS strategy. Further, whereas the CS strategy will read a memory state in constant time for any location in the trace (limited by disk IO speed and the size of the memory state), our strategy has to read the snapshot deltas preceding x and then use the deltas to reconstruct the memory state. However, while providing slower response times when compared to the pure CS strategy, our approach gives us much more control over how we balance preprocessing costs, disk utilization and query response times.

IV. Memory State Delta Tree

In the previous section we compared competing strategies for computing memory states from a trace. We also demonstrated how the BSD approach potentially offers a compromise between calculating memory state entirely on demand or precomputing state for each trace instruction. In this section we describe how the BSD strategy can be further improved by using a special type of B-tree to reconstruct memory state for i_x based on a traversal of that tree.

In Figure 1, we demonstrate how to represent a trace as a linear set of block snapshots. Taking this idea further, these snapshots can be aggregated into higher order blocks which cover even larger sections of the trace. This process can be repeatedly applied, until a tree is built whose root node covers the entire trace. This structure is what we term a memory state delta tree. Figure 2 depicts a partial delta tree for a 1M line trace. Each node contains a $startline$ and $endline$ along with a snapshot delta for the section of the trace from $startline$ to $endline$. From our discussion of the BSD, $endline$ here effectively corresponds to the last line of a trace block. The overall design of this data structure is to start with the root node containing the memory state difference between the first and last line in the trace. The next level in the tree increases the granularity (e.g., between the 1st and 1000th line, then the 1001st line to 2000th, and so on). Additional levels of granularity are introduced until nodes are created that correspond to the differences between adjacent lines. In this example, tree fan-out (the number of child nodes per parent node) is arbitrarily set to 1000 nodes. Tree fan-out is equivalent to the $blocksize$ parameter in the BSD strategy and performs a similar function controlling the structure of the delta tree.

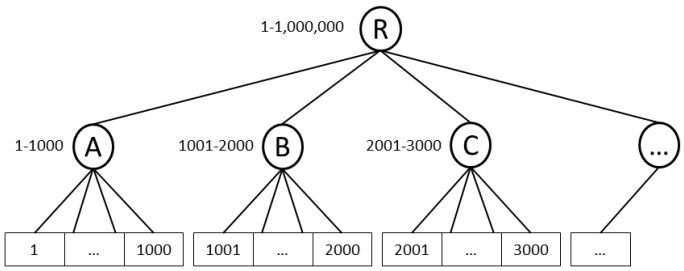

Fig. 2. Delta Tree

Reconstructing memory state at instruction i_x using a delta tree requires a simple traversal of the tree. Given a delta tree representing the cached memory state of a program and asked to reconstruct the memory state at x, we do the following traversal. Starting at the root, if $x = endline$, then we have found the node containing the last snapshot delta we require to reconstruct the memory state at x. If $x > endline$, then we know that node contains a snapshot delta that we need to reconstruct the state at x, so we traverse to the next sibling and continue our traversal. If $x < endline$, we traverse to that node's first child and so on until we find a node where $x = endline$.

We illustrate this with a simple example. If given the delta tree in Figure 2 and asked to reconstruct the memory state for instruction record i_x where $x = 10$, we would do the following traversal. Starting at the root R, we check if $x <= endline$. As $10 <= 1000000$, we know that R's snapshot is for a later point in the trace, so we investigate its first child, in this case A. Again we check if $x <= endline$ and again the answer is yes ($10 <= 1000$). However, when we visit A's first child, we see that $10 <= 1$ is false. We now know that this node contains a snapshot delta relevant to our query, so we save that node's delta and then traverse to the next sibling until we reach a node where $endline = 10$. Having reached a node where $x = endline$, we save that node's delta, and as we now know we have visited all nodes we need to reconstruct the memory state at instruction record i_x, we stop our traversal of the delta tree.

As another example, for $x = 2002$ the sequence of nodes visited would be as follows: $R, A, B, C, C_{2001}, C_{2002}$. Of these nodes, only A, B, C_{2001}, C_{2002} contain snapshot deltas required to reconstruct the memory state at $x = 2002$. In this example, A and B's children are not visited as A and B already contain the aggregated deltas of all their child nodes. This summarization property of child nodes is a significant feature of the delta tree, effectively caching delta computation for sets of trace instruction records and significantly reducing the computations required to reconstruct a memory state. In the worst case scenario, the maximum number of nodes needed to reconstruct a program's memory state is $((blocksize - 1)treedepth)$. And as can be seen in the example in Figure 2, a tree depth of 3 and $blocksize$ of 1000 can represent a 1M line trace, while a tree depth of 4 can represent a 1B line trace. The number of trace instructions

that can be represented by a delta tree of $treedepth$ is given by $(blocksize)^{treedepth-1}$.

The space required to store a delta tree is based on the total number and size of the snapshot deltas that are stored in each level of the tree and the number of memory address-value pairs contained in each. To determine this we first calculate the total size of all deltas stored in a delta tree as:

$$deltasize = (blocksize^{depth-1})depth = n(depth)$$

For example, if we consider a 1M line trace delta tree as per Figure 2. The bottom most layer of this tree contains 1M snapshot deltas, each containing a single memory reference, corresponding to the 1M trace instructions. The layer of nodes above this contains 1000 snapshot deltas each representing 1000 trace lines. The root layer then contains 1 snapshot delta containing 1M references. Therefore, the total number of memory references stored in the delta tree is 3M, 1M per level.

If we assume that each memory reference contains an 8-byte address and an 8-byte value, then the total disk space required to store all the deltas for a 1M trace file is 48MB. The space requirement for the delta tree grows as the delta tree's depth increases corresponding to this formula:

$$n(depth) = n\lceil log_{blocksize}(n)\rceil$$

For example a 1 billion line trace with $depth = 4$ and a $blocksize = 1000$ would require 64GB. Similarly, a 1 trillion line trace with a delta tree $depth = 5$ and a $blocksize = 1000$ would require 80TBs to store the worst case delta tree. While these are obviously very large storage requirements when considered in terms of a single workstation use case, they become quite modest if considered in terms of a distributed computing infrastructure.

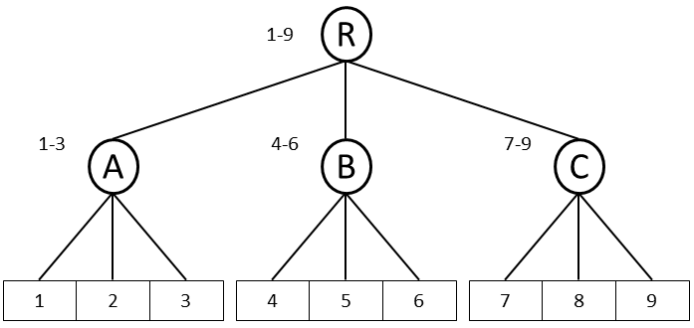

Fig. 3. A Small Delta Tree

V. Implementing the Delta Tree

We could conceivably implement a delta tree as an in-memory data structure for smaller traces, however, as the trace size increases, we would quickly exceed the memory capacity of the typical analyst workstation with just one delta tree. Further, as the fuzzing process can potentially generate hundreds or thousands of traces for a single program, we require a method that allows us to store and quickly query thousands of multi-gigabyte delta trees. The scale of the dataset requires a reconceptualizing of the trace analysis problem and the generation and querying of delta trees from a local process on a single analyst's workstation to a remote service on a distributed computing infrastructure. In this section we describe how we implemented delta trees using an off-the-shelf relational DBMS. Our rationale for choosing a database to implement a delta tree was influenced by the fact that the Atlantis trace analysis tool is already able to query traces stored in a database but also by a desire to allow multiple users to simultaneously access the data set. While we ultimately see delta trees as being implemented as a custom distributed computing binary data structure, a relational database serves as a conservative stepping stone to a truly distributed architecture.

To encode our delta tree in a database, we create a single database record for each node in the tree, storing the corresponding delta snapshot for a node as a blob object on the record. Reconstructing the memory state for a particular instruction record in the trace is then a process of querying the database for the relevant set of records (nodes) and combining the blobs (delta snapshots). To allow us to perform this query efficiently, when constructing the database, we label each node in our delta tree with a unique sequential identifier. The nodes are labeled incrementally from left to right, bottom to top, using a zero-based index. For example, to represent the tree in Figure 3, in the database we would assign the leaf nodes ids 0 through 8, nodes in level 2 would be assigned 9 to 11, and the root node would be assigned the id 12, resulting in the record structure in Figure 4. Conceptually we also label each node with a level-position pair that corresponds to its level in the tree and its index in that level of the delta tree.

Using this labeling scheme, querying the database for a trace instruction record can then be decomposed into a set of max $treedepth$ sequential queries for the relevant set of records (one query per tree level). For example, to query a database corresponding to Figure 4 for the snapshots required to reconstruct memory state at instruction record 5 in the trace would require two queries for the records in the positions $\{(2,0)\}$ and $\{(3,3)\ (3,4)\}$, while a query for trace instruction record 7 would require queries for $\{(2,0),\ (2,1)\}$ and $\{(3,6)\}$. If we know the $branchfactor$ of the tree, which in this case corresponds to $blocksize$, we then convert these level position pairs into their respective record ids, which can be done using the following formula:

$$id(level, position) = \left(\sum_{i=1}^{(treeheight-level)} branchfactor^{(treeheight-i)}\right) + position$$

VI. Benchmarking

Closely related techniques for reconstructing memory state such as [12] and [18] are designed to reconstruct the state of either single memory locations or subsets of memory. As our

technique reconstructs the entire memory state of the program under study, these techniques are not directly comparable. Therefore, in order to evaluate our memory reconstruction technique we chose a benchmarking approach instead of a direct comparison.

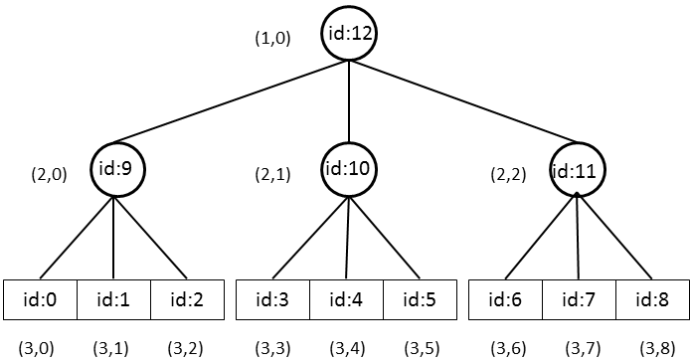

Fig. 4. Delta Tree Database Node Labeling

To benchmark our approach we performed an analysis of both the preprocessing time required to construct the memory state database and the time required to perform queries for different locations in the trace. We performed our analyses for a selection of trace sizes ranging from 600K to 60M lines (50MB to 5GB trace file sizes). All benchmarking was performed on a typical analyst workstation consisting of an Intel core i7 quad core processor running at 2.8GHz with 8GB of RAM and a 1TB 7200RPM hard disk. Our memory state database was implemented in a MySQL database. The database and the client-testing application were hosted on the same workstation to limit the impact of network latency on the test. When deployed to production, the memory state database could be hosted on a remote server and this network latency would have an impact on perceived query response times. However, because our technique allows us to page a relatively small quantity of memory state data at any one time, the impact on response times from network speed will be low and is not considered relevant to this test.

Table IV presents the results of our preprocessing analysis. From this table we see that the total time taken to process a 60M line trace file and store it in the database is just a little over 30 minutes. Given that this preprocessing step will be part of a larger automated batch trace generation process, this time is well within the requirements for our main use case. However, this analysis also shows that for smaller trace sizes our technique could conceivably be used by an analyst in a more interactive scenario, processing and indexing a 6M line trace file on demand in only a few minutes. Table IV also shows that Indexing Time, the amount of time taken to create the database index needed to support our record range queries, is a significant component of the total preprocessing time. As trace size increases, approximately 50% of the total preprocessing time is consumed by creating the index. If we divide the number of seconds taken to perform preprocessing by the number of trace instruction records (trace lines), we get

TABLE IV
PREPROCESSING BENCHMARK

Trace Size (lines)	600K	1.2M	6M	12M	60M
File Size (MB)	50MB	100MB	500MB	1GB	5GB
DB Insert Time (sec)	9	19	97	208	1,077
Indexing Time (sec)	2	4	70	141	885
Total (sec)	11	23	167	349	1,962

TABLE V
MEMORY SNAPSHOT DELTA DATABASE SIZE

File Size Data	Data (MB)	Index (MB)	Total (MB)
50MB	37	19	56
100MB	77	38	115
500MB	381	191	572
1GB	812	382	1,194
5GB	4,116	1,943	6,059

the number of seconds taken to process a single instruction record. Figure 5 shows that for database inserts, this figure is approximately constant over the range of trace file sizes. However, as the number of trace lines increases, we see a corresponding increase in the indexing time and the total processing time. This increase in indexing time is to be expected given the increase in the number of deltas that need to be indexed as trace file size grows. Interestingly, a hundred-fold increase in trace file size (from 600k to 60M trace lines) only results in an approximate three-fold increase in the index creation time per trace line (from 0.000005 to 0.000015). Table V presents an analysis of the memory snapshot data and index sizes for the different trace file sizes. As we can see, the space required to store the memory snapshot data in a database is not insignificant, however, it is consistent with the trace file size and well within our use case.

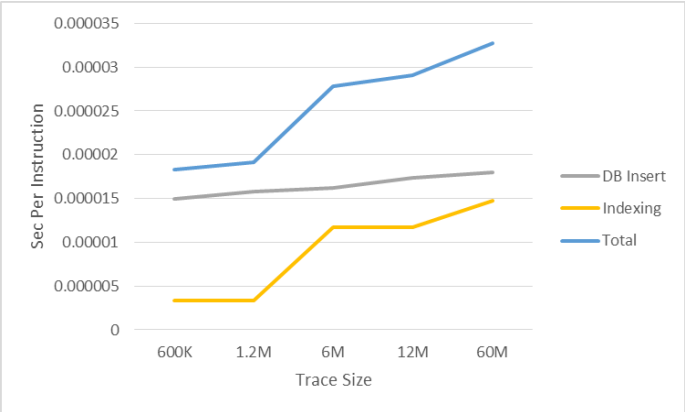

Fig. 5. Processing Time Per Instruction

When preforming the query response analysis, and in an attempt to accurately characterize the performance of our technique, we tested memory state reconstruction performance for a range of different size trace files and at multiple points in those files. Table VI presents the results of our query response analysis. These results were attained by making memory reference queries from a Java test harness program

TABLE VI
QUERY RESPONSE BENCHMARK

Trace Size (lines)	600K	1.2M	6M	12M	60M
File Size (MB)	50MB	100MB	500MB	1GB	5GB
Line at 10% (ms)	4.67	7.27	20.80	14.07	27.07
Line at 20% (ms)	5.70	9.40	14.03	18.70	32.73
Line at 30% (ms)	6.77	13.00	32.23	22.90	41.60
Line at 40% (ms)	8.83	16.63	19.77	27.03	42.67
Line at 50% (ms)	11.47	18.70	23.40	27.57	62.93
Line at 60% (ms)	13.50	24.47	23.43	18.20	55.67
Line at 70% (ms)	15.10	26.50	14.53	23.40	67.07
Line at 80% (ms)	15.60	29.13	27.57	30.67	78.53
Line at 90% (ms)	17.67	10.40	20.30	34.83	90.47
Line at 100% (ms)	19.23	13.53	22.87	41.10	88.90

to our MySQL database. Across the five trace file sizes, ten different target lines were queried against. The target lines were at every 10% increment of the number of lines in the trace file. To ensure that we were not hitting best case scenarios for the database index, target lines were not directly queried; 30 samples were taken for each query, adjusting the target line randomly by an amount equal to the target line, +/- rand(-0.5, 0.5) * 1000 lines. Timing was begun just prior to computation of the query arguments, and ended after return of the query results to the Java program. Timing is thus closer to real world interaction, including same-machine transmission time. As can been seen in Table VI, the response times for all queries in all trace files were significantly under our target response time of 500ms. The median response time for the 600k line trace file was 12.4ms, while the median response time for the 60M line trace was 59.3ms, representing a less than 5x increase in response time for a 100x increase in trace file size.

VII. DISCUSSION AND FUTURE WORK

The goal of this research is to investigate methods for reconstructing memory state from very large assembly traces so as to allow analysts to interactively query the memory state of a program in real time. Our analysis of the problem of memory state reconstruction and the memory state delta tree memory reconstruction approach that we present in this paper represent novel and significant contributions to the state of the art in assembly trace analysis techniques. We also present an analysis of the computational and storage costs associated with the delta tree approach demonstrating its practicality. Finally, we present a method where we implement the memory state delta tree using a commercial off-the-shelf DBMS allowing us to benchmark our approach and to evaluate its performance against our theoretical performance calculations.

The benchmarks presented in the previous section demonstrate that our delta tree technique for reconstructing memory state for very large traces delivers storage and query response performance in line with our theoretical calculations and significantly below our target average response time of 500ms. While we do see increases in the average response time as the size of the trace file increases, the rate of increase in response time is significantly lower than the rate at which the trace file itself increases. While this satisfies our current response time goal (for extremely large files, such as a billion line plus trace), there is a possibility that our technique as described would not be able to maintain these response times.

In future work we intend to investigate strategies for distributing memory state queries over multiple server instances, as well as methods for distributed indexing of trace files which should offer dramatic reductions in indexing time for large trace files. We also intend to investigate methods for performing higher-level analyses over the memory state delta tree which may be useful to analysts. For example, using the delta tree we could quite easily calculate all the values assigned to a memory location over the course of an execution trace, allowing an analyst to ask questions such as "show all lines in the trace where this address is assigned this value" and receive responses in real time. Finally, the memory state reconstruction approach presented in this paper makes a powerful new tool available to security analysts performing exploitability analysis, one which dramatically reduces the time required to get answers to questions compared to previous tools. To assess what impact this has and to gather requirements for other higher-level types of analyses that might now become practical, we intend to perform qualitative and quantitative evaluations of the impact of Atlantis and the delta tree memory state reconstruction approach on security analysts' work practices by studying those analysts while they use these tools.

VIII. RELATED WORK

There is a large body of work in dynamic analysis for reverse engineering and performance analysis that studies methods and tools for aiding software engineers in understanding the dynamic behavior of programs and systems [6], [23]. However, many tools from these fields tend to assume the availability of high-level source code and use extra structural information present in the source (package names, meaningful function, variable names, objects etc.) to assist in visualizing, navigating and understanding a program's trace and memory state. For analysts performing binary exploitability analysis, much of this higher-level information may not be available and so these tools and the visualization and summarization techniques employed by them are not usually readily applicable.

Reverse debugging, also called replay debugging [13], [17], is another closely related field of research and practice. Reverse debuggers record the history of instructions executed by a program while debugging to allow users to step forwards and backwards in the program's instruction history. Two popular debuggers, Microsoft's IntelliTrace [14] and GDB [8], both allow users to pause, rewind and resume programs while debugging and to interrogate the memory state of the debugged program while stepping through it. Compared to the trace analysis tools discussed in this paper reverse debuggers differ principally in their intended use case. Reverse debuggers are primarily intended to support development and debugging as well as some reverse engineering tasks. Reverse debuggers can be classified as supporting interactive online analyses, that

is, they are typically used to perform a single analysis or answer a single question at a time. On the other hand, trace analysis tools support offline analysis techniques, allowing users to jump to arbitrary points in the trace, and supporting queries and analyses defined over either the entire history of the program's execution, such as reconstructing the entire programs memory state.

Typically, memory visualization tools are designed to assist programmers in understanding how memory is being used by their program [1], [19]. Principally, these tools are used by developers to perform performance optimization of their programs. However, they can also be used for developer education and training. The Memory Trace Visualizer [3] typifies these tools providing animated views of memory accesses performed by a program as the user steps through the program's instructions. These views help the user identify potential performance optimizations that might result from using improved memory access patterns. These tools, like reverse debuggers, require a traversal of intervening program states to allow users to jump from one part of a program's execution to another. As such these tools are principally applicable for performing online analyses. However, the visualizations used by these types of tools do present interesting possible opportunities for improving how memory state is presented and visualized in trace analysis tools like Atlantis.

More recently, with increasing focus on low-level software security, researchers and organizations have started building tools specifically for binary security analysis that work directly with the program's executable at the assembly level. For example the BitBlaze [20] and BAP [2] platforms provide a comprehensive set of tools for performing static and dynamic analysis of binaries. The TEMU tool (part of BitBlaze) and Intel's Pin [11] use different approaches to generate execution traces, however, both can record effects on the state of a program's memory and registers at each instruction. Dynamic analysis platforms like TEMU and Pin are used to generate the types of traces that are then analyzed by tools like Atlantis.

The popular disassembler and debugger IDA Pro provides a 'Trace Replayer' [10] feature that allows users to enter into a pseudo-debugger mode and replay traces of programs captured while being debugged. Along with the disassembly of the instructions executed, IDA Pro also allows the user to observe the effects of instructions on register values based on linear reconstruction of the register state. Similarly, the Malfare Plugin [7], a prototype plugin developed for IDA Pro, uses traces generated by the TEMU dynamic analysis platform to allow users to step back through a program trace within the IDA Pro interface. Like the work we present in this paper, Malfare also allows users to reconstruct the memory and register state of the program at any point the trace, however, Malfare only reconstructs the memory state on demand in a linear fashion when requested by the user, requiring a recalculation of the state each time a request is made.

TaintBochs [4] is a whole system taint analysis framework designed to help identify how sensitive information like passwords and credit card information, once read into memory, propagates through a system and how long it resides in memory. TaintBochs allows users to reconstruct the memory state of the system at any point in a program's execution by taking a full snapshot of the state of the memory of the system when logging beings. Using this snapshot and an execution log that records all memory references, TaintBochs is then able to reconstruct memory state by applying instructions from the execution log to the initial snapshot until the desired point in the execution log is reached. The memory reconstruction approach taken by TaintBochs could be described as an example of the LR method discussed in section 3.

Tralfamadore [12] is a platform for performing whole system execution trace recording and offline analysis. Like Atlantis it aims to decouple trace generation and indexing from the analysis stage, allowing analysts to execute multiple queries against a cached trace interactively. To accomplish this, Tralfamadore indexes various aspects of the trace in an attempt to speed up query response times by reducing the amount of trace that needs to be traversed to evaluate a query. For example, a query for all the values of a memory address would query an index for memory allocations, identify the trace lines that affect that memory location and then do a traversal of those trace lines to reconstruct the memory state of that address over the lifetime of the execution. This approach allows queries for slices of memory state to be computed relatively efficiently (on the order of seconds). The memory state indexing and reconstruction approach that we present in this paper differs in that our index actually stores snapshots of the entire state of the program under analysis. This approach allows us to compute the entire state of memory of a program at any point in a trace without having to do a traversal of the trace. Both approaches have merits and should be seen as complimentary; the Tralfamadore approach requires much smaller index sizes but will result in slower response times and will only support queries for a subset of the memory state of the program. The approach presented in this paper requires larger index sizes but will be able to respond to queries faster and better support queries for the entire memory state of the program.

In [18], the authors present STIQ (Summarized Trace Indexing and Querying), a disk-based reverse debugging approach that allows users to capture, index and interactively navigate large java execution traces while also allowing users to query the state of memory locations at arbitrary points in the trace. Like the approach we present in this paper, STIQ uses a snapshotting technique to segment an execution trace and allow efficient reconstruction of control flow and program state. However, while the two approaches are outwardly similar, the STIQ approach differs from the approach presented in this paper in several important ways. Firstly, STIQ is optimized so that the indexing and tracing operations can be done in relatively interactive time scales to allow for developers performing debugging/development use cases. The exploitability analysis use case our technique is design to support does not require that trace generation and indexing happen in interactive time scales which allows higher fidelity

traces to be captured and more complex indexes to be created. Secondly, the STIQ approach to memory state reconstruction is similar to that of the Tralfamadore approach, using indexes to allow efficient linear reconstruction of the state of a single memory location at a time. This approach differs significantly from the type of memory state reconstruction discussed in this paper where we reconstruct the entire memory state of the program under study by combining snapshot deltas and by performing some limited linear reconstruction. This capability is required by the exploitability analysis use case and while our approach obviously requires much more storage space when compared to these other approaches, it does allow us to reconstruct the entire memory state of the program under study in not much more time than other approaches reconstruct the state of a single memory location.

IX. Conclusion

In our previous work [22], we reported on a first-of-its-kind field study of the work practices of software security engineers where we identified the tools and processes used, as well as the unique constraints and challenges those engineers face. Based on that work we developed the Atlantis trace analysis environment [5] to assist engineers in performing exploitability analysis of multi-gigabyte assembly execution traces. In this paper, we present a novel preprocessing method for reconstructing the memory state of a program from its execution trace. We show that our technique allows security analysts to perform real-time queries about the memory state of a program at any point in its execution for very large traces. This new capability will, we hope, dramatically decrease the time spent by analysts in calculating memory state and improve the trace analysis work flow. In future work we intend to perform an empirical evaluation of how our tools, and this technique in particular, impacts security analyst work practices. We also intend to investigate methods for implementing our memory state reconstruction approach using a distributed computing infrastructure.

X. Acknowledgments

We wish to thank Defence Research and Development Canada and Cassandra Petrachenko for assisting with this research. This research is funded through NSERC grant DNDPJ 380607-09 and DRDC Valcartier.

References

[1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. Heapviz: interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 53–62, New York, NY, USA, 2010. ACM.

[2] David Brumley, Ivan Jager, Thanassis Avgerinos, and EdwardJ. Schwartz. Bap: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 463–469. Springer Berlin Heidelberg, 2011.

[3] A. N. M. Imroz Choudhury, Kristin C. Potter, and Steven G. Parker. Interactive visualization for memory reference traces. In *Proceedings of the 10th Joint Eurographics / IEEE - VGTC conference on Visualization*, EuroVis'08, pages 815–822, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[4] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[5] Brendan Cleary, Margaret-Anne Storey, Laura Chan, Martin Salois, and Frederic Painchaud. Atlantis - assembly trace analysis environment. *Reverse Engineering, Working Conference on*, 0:505–506, 2012.

[6] Wim De Pauw and Steve Heisig. Zinsight: a visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 143–152, New York, NY, USA, 2010. ACM.

[7] Dominic Fischer and Daniel Jordi. Malfare plugin for IDA Pro. Available online: https://www.hex-rays.com/contests/2011/index.shtml. Last access: 6/7/2013, June 2013.

[8] GNU. Gdb: The gnu project debugger. Available online: http://www.gnu.org/software/gdb/. Last access: 6/7/2013, June 2013.

[9] Toms Hardware. 2012 ssd benchmarks. Available online: http://www.tomshardware.com/charts/ssd-charts-2012/AS-SSD-Sequential-Write,2783.html. Last access: 6/28/2013, June 2013.

[10] HexRays. IDA Pro: The trace replayer. Available online: http://www.hexblog.com/?p=669. Last access: 6/7/2013, June 2013.

[11] Intel. Pin - a dynamic binary instrumentation tool. Available online: http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. Last access: 6/7/2013, June 2013.

[12] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution mining. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 145–158, New York, NY, USA, 2012. ACM.

[13] Bil Lewis and Mireille Ducasse. Using events to debug java programs backwards in time. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 96–97, New York, NY, USA, 2003. ACM.

[14] Justin Marks. Debugging applications with intellitrace. *MSDN magazine*, page 36, 2010.

[15] Microsoft. "!exploitable crash analyzer - msec debugger extensions. Available online: http://msecdbg.codeplex.com. Last access: 6/4/2012.

[16] Jakob. Nielsen. *Usability engineering*. Elsevier, 1994.

[17] G. Pothier and E. Tanter. Back to the future: Omniscient debugging. *Software, IEEE*, 26(6):78–85, 2009.

[18] G Pothier and E Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *ECOOP 2011 Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 558–582. Springer Berlin Heidelberg, 2011.

[19] George G. Robertson, Trishul Chilimbi, and Bongshin Lee. Allocray: memory allocation visualization for unmanaged languages. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 43–52, New York, NY, USA, 2010. ACM.

[20] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[21] M. Sutton, A. Greene, and P. Amin. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[22] C. Treude, F. Figueira Filho, M.-A. Storey, and M. Salois. An exploratory study of software reverse engineering in a security context. In *18th Working Conference on Reverse Engineering (WCRE 2011)*, pages 184–188, 2011.

[23] Jonas Trümper, Johannes Bohnet, and Jürgen Döllner. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 133–142, New York, NY, USA, 2010. ACM.