Eidetic Systems

David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen
University of Michigan

Abstract

The vast majority of state produced by a typical computer is generated, consumed, then lost forever. We argue that a computer system should instead provide the ability to recall any past state that existed on the computer, and further, that it should be able to provide the lineage of any byte in a current or past state. We call a system with this ability an *eidetic computer system*. To preserve all prior state efficiently, we observe and leverage the synergy between deterministic replay and information flow. By dividing the system into groups of replaying processes and tracking dependencies among groups, we enable the analysis of information flow among groups, make it possible for one group to regenerate the data needed by another, and permit the replay of subsets of processes rather than of the entire system. We use model-based compression and deduplicated file recording to reduce the space overhead of deterministic replay. We also develop a variety of linkage functions to analyze the lineage of state, and we apply these functions via retrospective binary analysis. In this paper we present Arnold, the first practical eidetic computing platform. Preliminary data from several weeks of continuous use on our workstations shows that Arnold's storage requirements for 4 or more years of usage can be satisfied by adding a 4 TB hard drive to the system.[1] Further, the performance overhead on almost all workloads we measured was under 8%. We show that Arnold can reconstruct prior state and answer lineage queries, including backward queries (on what did this item depend?) and forward queries (what other state did this item affect?).

1 Introduction

The vast majority of state produced by a typical computer is generated, consumed, then lost forever. Lost state includes process address spaces, deleted files, interprocess communication, and input received from the network. With lost state comes lost value: users cannot recover detailed information about past computations that would be useful for auditing, forensics, debugging, error tracking, and many other purposes.

Prior approaches try to retain some of this information via a variety of techniques, such as file backup, packet logging, and process checkpointing, but these approaches preserve only the subset of information that someone anticipates may be useful. A more comprehensive approach is needed: one that preserves the values and lineage of *all* state that has ever existed on the system. We call such a system an *eidetic computer system*.

An eidetic computer system can recall any past state that existed on that computer, including all versions of all files, the memory and register state of processes, interprocess communication, and network input. Further, an eidetic computer system can explain the lineage of each byte of current and past state.

Lineage describes *how* state was derived. With such information, the user of an eidetic system can often infer *why* the data was derived. For instance, a colleague might point out to a user that a citation in a paper draft is incorrect. Using an eidetic system, the user could trace back from the binary document through all the steps used to create that document and recreate the browser screen displaying the Web page from which the data was derived. On seeing that Web page, the user would realize that he cited the wrong paper from a conference session. The user could then trace forward from that mistake and reveal all current documents and data that reflect the mistake, as well as any external output (e.g., e-mail) containing mistaken information.

Or consider an example in which someone runs a malicious application on a shared computer. The malicious program exploits a privilege escalation vulnerability, gives itself privileged access, and installs a backdoor for future access. An eidetic system could trace forward from the malicious program, trace through the priv-

[1] Currently, a 4 TB drive can be purchased for approximately $150.

ilege escalation vulnerability, and determine that the malicious software installed a backdoor. The system could then trace any future executions of the vulnerable program and determine if the backdoor was ever used, and exactly what was done by the attacker during the vulnerable window. In these and similar examples, recall and lineage are tightly coupled; they are useful in isolation but much more powerful when combined.

In this paper, we describe an eidetic system called Arnold that provides the above properties for personal computers and workstations with reasonable storage requirements and runtime overheads. The key technologies that enable Arnold to provide the properties of an eidetic system efficiently are deterministic record and replay [10], model-based compression, deduplicated file recording, operating system tracking of information flow between processes [23], and retrospective binary analysis of information flow within processes [11, 35].

Arnold uses deterministic record and replay to efficiently reproduce past computations. Reproducing past computations enables Arnold to recall any state and to track the lineage of that state within a replaying entity. Arnold uses numerous optimizations to reduce the amount of data that must be recorded. As a result, the log data required for years of operation of a personal computer or workstation can fit on a commodity hard drive.

To avoid the need to replay the entire system to recover any state, Arnold divides the system into units, called *replay groups*, that can be replayed independently. To track information flow between replay groups, Arnold records dependency information for each communication between replay groups, forming a *dependency graph*. In addition to enabling information flow to be tracked across groups, the dependency graph also allows Arnold to treat as a cache the log of data sent between groups. To conserve space, Arnold can discard this data and regenerate it later by replaying the group that produced it, a technique we call *cooperative replay*.

To analyze lineage within a replay group, Arnold uses retrospective binary analysis, in which it deterministically reexecutes the processes within the group and tracks the relationships between inputs and outputs. Different *linkage functions* may be used to define dependencies according to how the lineage analysis will be used. Arnold defers the choice of linkage function to the time of the query. This preserves flexibility and enables it to answer lineage queries that were not anticipated during the original execution. It also moves the overhead of analysis from original execution to the time of query.

Putting these pieces together, Arnold can answer both backward queries (where did this particular state come from?) and forward queries (what outputs and current state are derived from this prior state?). It does so by following the dependency graph, reexecuting the process groups to learn how their inputs map to their outputs, and querying the graph to learn how group outputs map to the inputs of other groups.

Underlying Arnold's design is the observation that deterministic replay and information flow are synergistic. Recording information flow among processes saves storage space by eliminating the need to record the data sent between processes. Deterministic replay makes it possible to reproduce any transient state of a prior process execution. This makes it possible to perform information flow queries over that state that were not imagined at the time the process executed. It also provides the ability to defer the work of tracking information flow within processes until the results are needed.

We have run Arnold continuously on our workstations for several weeks. Our results show that its storage requirements for 4 or more years of operation could be satisfied by adding a $150 4 TB hard drive. On almost all benchmarks we ran, Arnold's performance overhead is less than 8%. We also report on several case studies in which we use Arnold to reproduce past state and trace lineage over many applications and workflows.

2 Related work

Arnold draws upon prior research from many areas. Many systems have sought to save some prior state in a system. For example, versioning file systems [39, 44, 49] store regular snapshots of file state; process checkpointing systems [40] store snapshots of running processes; and systems like DejaView snapshot both processes and files [24]. These systems store only a subset of the state in a system, and they take checkpoints only at coarse-grained points in time to reduce storage usage. Checkpoints are insufficient to reproduce computation or to track intra-process lineage. In contrast, Arnold can reproduce all state and computation in a system at the granularity of individual instructions.

Arnold uses deterministic record and replay to reproduce all state and computation in a system. Deterministic replay for uniprocessors is a mature technology, and implementations exist in both software [54, 10, 14, 17, 36, 43, 46] and hardware [6, 34, 53, 21, 30, 33, 51]. Making deterministic replay efficient on multi-processors is an ongoing research challenge [15, 25, 50, 3, 38, 52, 55, 12, 26], as is making execution on multiprocessors deterministic [8, 9, 13, 28, 37]. We deterministically replay a multiprocessor system by recording synchronization operations and instrumenting races, but this is not a focus of this research. Instead, we focus on applying deterministic record and replay to build an eidetic system and on reducing the storage overhead for long-term use through techniques such as model-based compression.

Checkpointing and rollback-recovery have often been

used to restore past state [16]. However, the focus of most prior work has been to tolerate failures by reproducing the latest correct state, rather than to restore any past state as in eidetic systems.

Prior research has examined how to track the lineage of data, either within a process [35] or between processes [23, 22, 18]. Provenance-aware storage systems [31, 32] annotate file data with causal history to capture the relationship between processes and files. Arnold tracks lineage within a process, between processes, and between files and processes. Unlike prior systems, Arnold tracks comprehensive lineage for arbitrary executables and non-deterministic programs.

Other projects have examined how to index and query prior system state. DejaView [24] indexes and provides a query interface for prior information that is displayed on the screen or available through the accessibility API. Tralfamadore traces the execution of a system and provides mechanisms and components to analyze that trace [27]. Arnold provides the ability to reproduce all state and track its lineage across arbitrary computations.

Our technique to regenerate inter-group communication via replay trades storage for recomputation. Other projects have made a similar tradeoff in different domains [19, 2, 7, 47]. For example, Nectar [19] trades storage for computation in data-parallel cloud environments and supports recomputation of storage results from inputs and memoization of partial and full computations. While Nectar is restricted to DryadLINQ applications, which are both deterministic and functional, Arnold provides these and other benefits for general-purpose computation.

3 Design goals

The design of Arnold was guided by several goals. First, we wanted to support the widest possible range of queries about user-level state and the lineage of that state. Arnold reproduces and tracks the lineage of state of all user-level processes at the level of the instruction set architecture. We wanted to support queries (Section 4.8) about backward lineage (what influenced this data?) and forward lineage (what did this data influence?) both within a replay group (Section 4.6) and between replay groups (Section 4.5). We also wanted to support queries not anticipated at the time of recording, which we accomplish via retrospective binary analysis (Section 4.6).

Second, we wanted to minimize the time and space overhead of recording, since we intend for Arnold to continuously record computer usage. We wanted the time overhead of recording to be low enough to support interactive workloads and the space overhead to be small enough to record several years of execution of workstations and personal computers on a commodity hard drive. We reduce the time overhead of recording through deterministic record and replay (Section 4.1) and retrospective binary analysis (Section 4.6). We reduce space overhead through techniques such as model-based compression (Section 4.2), deduplicated file recording (Section 4.3), and cooperative replay (Section 4.4).

Third, we wanted to reduce the cost of answering queries by not requiring the reexecution of processes unrelated to the state being queried. We accomplish this by dividing the system into multiple replay groups, each of which can be replayed independently. To preserve lineage between replay groups, we track the dependencies cause by inter-group communication in a dependency graph (Section 4.5).

4 Design and implementation

4.1 Record and replay

Deterministic record and replay enables two important features of Arnold. First, it allows Arnold to efficiently reproduce the complete architectural state (register and address space) of user-level processes. Second, it allows Arnold to defer the work needed to track lineage from the time of execution to the time of querying [11].

To enable reproduction of all architectural state, Arnold records and replays execution at the level of processes. Our modified Linux kernel records all nondeterministic data that enters a process: the order, return values, and memory addresses modified by a system call; the timing and values of received signals; and the results of querying the system time.

Dealing with multiple threads/processes that write-share memory requires special care. Record and replaying individual threads/processes would shrink the scope of replay needed to answer a query, but this would require Arnold to record all nondeterministic reads of shared memory. Instead, Arnold records all threads/processes that share memory as a single *replay group*, then seeks to replay the interleavings of events from the replay group deterministically.

To enable deterministic replay of a replay group, Arnold records all synchronization operations and atomic hardware instructions (such as `atomic_inc`, or `atomic_dec_and_test`). A modified version of `libc` logs the order and memory addresses of synchronization operations between threads, including low-level atomic instructions and high-level synchronization operations such as `pthread_lock`. Such logging inserts an additional two atomic instructions for each event logged (to order the start and end of the operation). In the absence of data races, this information is sufficient to faithfully replay the recorded execution of a replay group involv-

ing multiple threads or processes—each replayed thread will execute the same sequence of instructions and system calls, observe the same values read, and produce the same results as during recording [42].

In the presence of data races, the replayed execution may diverge from the recorded one. We deal with programs with data races by identifying the races and adding additional instrumentation to eliminate them on subsequent runs. Veeraraghavan et al. [48] observed a synergy between deterministic replay and data race detection: if the only reason that a replayed execution may diverge from a recorded execution is the presence of a data race, then the replay system can act as a very efficient data-race detector. Arnold supports the ability to instrument and observe the execution of replayed recordings (Section 4.6), and we use this to run a standard vector-clock data race detector [41] when a replay divergence is detected. This is guaranteed to detect at least the first pair of racing instructions (it may also detect subsequent pairs). We then either statically instrument the code to record the outcome of the data race, or dynamically instrument the binary when it runs to cause the racing pair of instructions to trap to the kernel (via an `INT 3` instruction), where we record the order of the racing instructions. Static instrumentation is preferred since it is more efficient, but dynamic instrumentation allows us to support applications for which we do not have source code.

In practice, we have detected few data races that affect replay in the programs we run on our workstations. It has been relatively simple for a small team of users to add the necessary instrumentation to record these instances. Interestingly, many of the races we found were already documented, for example by developers who ran ThreadSanitizer [45] or similar tools. Since races are very infrequent, we suspect that it should usually be possible to search through all possible interleavings of the racing instructions to find an interleaving that is indistinguishable from the recorded execution [3, 38].

When a process executes the `exec` system call, Arnold creates a new replay group (with a unique 64-bit identifier) consisting solely of that process. Arnold also saves a small checkpoint for the new group, which allows replay to begin from the creation of that process. The checkpoint consists of the arguments and environment variables passed to `exec`, other nondeterministic information used during the system call (e.g., seeds used to randomize address spaces), and a *reference* to the file containing the executable image—the image usually resides in a deduplicated file store described in Section 4.3.

Arnold creates new replay groups on `exec` rather than on `fork` because the initial address space at `exec` is more amenable to deduplication than the address space at the time of `fork`. It stores a *split record* that contains the unique identifier of the new replay group in the log of the replay group that performed the `exec`. Infrequently, two replay groups need to be merged (e.g., because they establish a write-shared memory segment). In such instances, Arnold merges the processes from one group into the other and inserts a *merge record* into their logs.

Arnold replays recorded execution on a per-group basis. It creates a new process from the group's checkpoint and deterministically reexecutes the process by supplying values from the group's log in lieu of performing any nondeterministic action. As additional threads and processes are created within the replay group, Arnold also replays those entities. Each process executes until it exits or the execution reaches a split record. Arnold can replay multiple groups concurrently—this allows it to parallelize lineage queries that span groups.

4.2 Reducing storage utilization

Arnold uses several optimizations to reduce the size of its replay logs. The first optimization is model-based compression. The order and results of many of the system calls and synchronization operations that Arnold logs are highly predictable. For instance, many system calls usually return zero (success); the `write` system call usually returns the number of bytes in the input buffer; and `pthread_cond_lock` usually returns a value specifying that the lock has been obtained. Arnold constructs a model for predictable operations and records only instances in which the returned data differs from the model. Thus, the log size used for each type of operation is proportional to the number of deviations, which can be much less than the number of executed operations.

Some operations such as `poll` exhibit considerable locality in the data they return (e.g., the set of ready file descriptors is often the same from call to call within a short window). For these operations, Arnold caches the most recent 8 values returned on both record and replay and replaces the actual values in the log with a small cache index (when the value hits in the cache) in order to save space. Arnold also uses model-based compression to reduce the amount of ordering information in the log. It predicts that there are no ordering constraints and no signals delivered between two successive logged operations, and records only when the execution deviates from the model.

After applying model-based compression, we determined that the most significant source of log usage on our systems was messages sent from the X server to applications. A small fraction of this data comes from user events (button presses, mouse movements, etc.). Most data consisted of responses to application requests. Since such responses included nondeterministic data such as identifiers and window properties, the responses needed to be recorded to faithfully replay each application.

We observed, however, that with the exception of ac-

tual user input, the behavior of the X server is mostly deterministic. Arnold avoids logging most data from the X server by using the X server to help regenerate data during replay. We insert a proxy between applications and the X server that records only a small subset of the data sent from the X server, such as identifiers and window properties generated nondeterministically by the X server. During replay, the application again connects to an X server via the proxy. The proxy translates the nondeterministic values, and the replay process generates GUI state using the live X server, but on a separate display. The proxy also inserts the recorded user events at the appropriate point in the stream. In combination with the proxy translation, the X server produces the same sequence of responses during the replayed execution as during recording. With deterministic X recording, Arnold can make the display of X windows visible during replay. As we will describe, this is useful for showing users application displays that correspond to the results of lineage queries and for allowing users to specify queries by clicking on recreations of windows displaying data they observed in the past. By recording only nondeterministic response values and user input, the proxy substantially reduces the amount of information in the logs of GUI applications.

After applying the above optimizations, we noticed that time queries constituted a substantial portion of the remaining log size. To reduce the amount of nondeterminism that needs to be logged, Arnold uses a *semi-deterministic clock*. The value returned by a semi-deterministic clock is guaranteed to be less than the real-time clock for the system, and within a specified delta. The default delta is 10 ms; it may be overridden by applications that need more accuracy. A replay group's semi-deterministic clock is incremented deterministically based on the number and type of logged operations (which is the same during both recording and replay). When the time is queried, Arnold reads the actual real-time clock. If the semi-deterministic clock is greater than or more than delta behind the real-time clock, Arnold returns the real-time clock value, sets the semi-deterministic clock equal to the real-time clock, and records the new value in the log. Thus, the amount of time query data in the log is proportional to the number of such resets rather than the total number of time queries; if Arnold usually predicts the clock value correctly, the amount of logged time data can be quite small.

Arnold ensures that observed semi-deterministic clock values are externally consistent. It is for this reason that the semi-deterministic clock must always be less than the real-time clock. If a recorded process sends a message to a non-recorded process, the receiver will always observe that the message arrived after it was sent. Further, if a recorded process receives data from or sends data to an entity outside the replay group, the group's semi-deterministic clock is set to match the real-time clock. Thus, the observed clock values are causally consistent both across all processes on the computer system and with respect to external entities.

Finally, Arnold compresses all log data with `gzip`. This is very effective in compressing some input, such as text. It also helps to compress applications that perform repetitive operations with similar results.

4.3 Copy-on-RAW file cache

Arnold records the file data read by a process so that data can be redelivered to the process during replay. Recording this data can take a substantial amount of log space, so Arnold optimizes how the read file data is stored by deduplicating it. This works particularly well when a file is read multiple times before being modified.

To deduplicate the read file data, Arnold saves a version of a file only on the first read after the file is written. Subsequent reads log only a reference to the saved version, along with the read offset and return code. We refer to this as *copy-on-RAW (read-after-write) recording*.

If another process opens the file for writing while a reading process is running, the reading process reverts back to recording the read values instead of the reference to the stored version (several optimizations are possible here, such as recording the file version on each `read` instead of `open`, or reexecuting reads and writes to files shared among processes in the same replay group.)

Arnold also uses the copy-on-RAW store for file `mmap` operations by mapping the stored file version into the process space on replay. If the mapped region is writable, Arnold creates a private temporary copy of the file version on replay; this allows the replayed process to change the file contents without affecting other replayed processes that reference the same file version.

Note that, with this design, the current version of all files is stored in the default file system (ext4 on our Ubuntu workstations). We chose this operation for efficiency; recording processes (the common usage case) go through the well-optimized file system and receive the best performance. Copy-on-RAW population of the file store can proceed asynchronously and not slow down the recording process too much unless large amounts of data being read exert memory pressure. The cost of this implementation is some double-buffering of current file data, which we could reduce in the future.

We were initially surprised because the size of Arnold's file store grew more slowly than expected on our workstations. On investigation, we realized this was due to an important difference between Arnold's file store and a versioning file system: Arnold's file store does not have to store data that is written but never read. Since Arnold is an eidetic system, it can, of course, recre-

ate this file data; however, it does so by replaying the process(es) that produced the data rather than by retrieving the data from the file system. In contrast, a versioning file system needs to store all file versions even if they are overwritten or deleted without being read.

Since Arnold can reproduce any current or past file version via replay, it is the logs of nondeterminism that are Arnold's truly persistent store [16]. We can thus treat Arnold's copy-on-RAW file store as a *cache*. The copy-on-RAW file store (and, in fact, all file system data) is simply a performance optimization that contains checkpoints of data that could be produced by replay. This reasoning led us to develop *cooperative replay*.

4.4 Cooperative replay

We normally think of replay groups as independent entities: we log their nondeterministic inputs during recording and reinsert these inputs during replay. Cooperative replay provides another option, which is to use one replay group to regenerate the data read by another. Cooperative replay allows us to treat the log of all interprocess communication (files, pipes, etc.) as a cache, whose records can be evicted when the cache is full and recovered when needed during replay.

Arnold uses cooperative replay to regenerate data read from files. During replay, if the requested data exists in the file system (because it is the current version of the file) or in the copy-on-RAW file cache, Arnold reads the data from one of those locations. If not, Arnold regenerates the data by replaying the replay group(s) that produced the data. Arnold stores information about the source of all file data in each read record—this includes the identifier of the replay group(s) and the system call(s) executed by the group(s) that produced the file data (Section 4.5). To regenerate the data, Arnold suspends the replay group requesting the data, replays the producing replay group(s), repopulates any data evicted from the file cache, and finally resumes the requesting replay group.

Cooperative replay may recurse in a depth-first manner. When replaying replay group A, Arnold may need to replay another replay group B to regenerate file data read by replay group A, and this may trigger the replay of a third replay group C, and so forth. The recursion will stop when a group can be replayed without depending on any other replay group.

As data flows forward, it creates a directed acyclic graph. While no cycles exist between nodes in the graph, Arnold may encounter a scenario where two replay groups depend on outputs of each other. In this scenario Arnold will alternate replaying each group until all dependencies are met.

4.5 Dependency graph

To support cooperative replay and track lineage across replay groups, Arnold maintains a logical graph of the data-flow dependencies between groups, which we refer to as the *dependency graph*. Nodes in the graph are <replay group id, system call id> tuples, where the second part of the tuple uniquely identifies a particular system call executed by a process in the replay group. Each edge in the graph is a bidirectional link between the system call that produced data and the set of one or more system calls that consumed that data. Thus, Arnold can determine the lineage of data across replay groups by tracing backward in time through the dependency graph, and it can determine what downstream values were influenced by particular data by tracing the lineage forward.

We first describe the operation of the dependency graph for file data. When a recorded process writes to a file, Arnold records which bytes were modified, along with the <replay group id, system call id> in a per-file B-tree indexed by the file offset. The root of each per-file B-tree is in turn indexed in a B-tree of all files; we refer to this collection of B-trees as the *filemap*. Arnold allocates a separate region on disk for the filemap; it reads pages on demand into a kernel cache in physical memory and evicts pages using an LRU algorithm. Pages are flushed asynchronously using the journal mechanisms of the underlying file system (ext4 in our current implementation). Thus, the filemap contains the lineage information for all current file data in the file system.

When a recorded process reads from a file, Arnold searches through the filemap to find which system call(s) wrote the bytes being read. It copies the tuples out of the filemap into the replay log of the reading process. Thus, the log contains sufficient data to answer backward lineage queries (how was the data read by this process produced?). In order to answer forward lineage queries, Arnold generates an index over the reverse linkages and stores it in a `sqlite` database. A daemon process asynchronously generates the index by incrementally scanning recent replay logs (replay is unnecessary because the data needed to generate the index is in the logs).

Arnold uses a similar process to record the lineage of other forms of IPC. For pipes and sockets, it keeps metadata for bytes written but not yet consumed in the kernel. For most pipes and sockets, there is a single writing process and a single reading process, and bytes are read in the order they are written. In this common case, Arnold reduces log size by only logging the identifier of the writing replay group. On a query, Arnold identifies the system call(s) that generated data by scanning the log of the writing record group. If there is more than one reader or writer, Arnold tracks the reads and writes on the pipe or socket in the same manner as for file system data.

Arnold also tracks lineage of the data passed from the

parent process to the child process during `exec`. This includes arguments, environment variables, and some miscellaneous data used during the `exec` system call.

Arnold does not record the lineage of data passed among processes via shared memory. Instead, Arnold tracks this lineage at query time by instrumenting the memory read and write instructions as described in the next section.

4.6 Intra-process lineage

Arnold uses Pin [29] binary instrumentation to analyze replayed executions and track the lineage of data within a replay group. We chose Pin because it is a flexible and well-documented tool; however, Pin can be slow, partially because it dynamically, rather than statically, inserts instrumentation into running binaries. Arnold avoids overhead during recording by only using Pin and analyzing intra-process lineage during replay.

While analysis tools such as Pin are typically invisible to the program they instrument, they are not transparent to the operating system: such tools insert new system calls, allocate additional memory, catch signals, etc. Without special care, these extra actions to support analysis will cause the replayed execution to diverge from the recorded execution. Arnold uses techniques from X-ray [4] to compensate for the divergences caused by analysis; for instance, it prevents Pin from allocating memory that will conflict with the replayed execution and it identifies system calls generated by Pin and executes them live rather than trying to supply nondeterministic values from the group's log.

Arnold traces lineage within a group by restoring the group's checkpoint and replaying the processes within the group with Pin dynamic analysis enabled. Pin tools determine which inputs to the group influenced which outputs according to a customizable *linkage function*.

There are many possible ways of defining lineage: one can say that an input influences an output only if the output is derived from the input via a series of copies, or one can consider other forms of data flow, or control flow, etc. The linkage function defines, for each type of processor instruction, which outputs of the instruction are influenced by which inputs. Each linkage is implemented as a Pin tool. Arnold provides several common linkage functions (and applications may define their own):

- **Copy.** An input of an instruction influences an output only if the instruction copies the value of the input to the output location (e.g., via a move instruction).
- **Data flow.** An input of an instruction influences an output if the instruction uses the input to calculate the value of the output (e.g., via an add instruction).
- **Index.** An input influences an output if the input is used to calculate the output or if the input is used as an index to load a value used to calculate the output (e.g., via an array or lookup table index).
- **Control flow.** This includes, in addition to index and data flow influence, the influence propagated via control flow as tracked using the algorithms developed by ConfAid [5].

The lineage data returned by each linkage function above is a superset of the preceding linkage functions. For example, if a group input and output are related via the data flow linkage, they are also related via the index and control flow linkages.

A lineage query for a group specifies a set of inputs, a set of outputs, and a linkage function. Inputs may be specified as a set of <system call id, byte range> tuples, where each tuple denotes a unique system call performed by the recorded group and the subset of input bytes to track for that call. Alternatively, the input may be specified as a class of input (e.g., all file data or all GUI events from the X server), or the query may simply track all inputs. Output is specified similarly.

Arnold uses taint tracking to derive intra-group lineage. When it sees a system call matching the input specification, it taints the requested bytes with unique identifiers as they are read into the process address space. As instructions execute, the Pin lineage tool propagates that taint among memory addresses and registers according to the linkage function. When Arnold sees an output matching the specification, it writes the taint of each output byte to a results file. Of course, each byte may be influenced by zero to many inputs.

4.7 User-propagated lineage

For interactive workflows, lineage may pass through the user of the system. For instance, she might view a Web page in a browser, then type text from that page into an editor. Arnold tracks such lineage by first identifying inputs and outputs that occurred at approximately the same time, then using fuzzy string matching to look for contextual linkages among those inputs and outputs.

Arnold identifies user-generated inputs with a Pin tool that runs on a replayed execution. The tool identifies channels corresponding to user input: sockets used to communicate with the X server (for GUI input), the terminal device (for text input), and network sockets connecting to well-known ports associated with user input (e.g., the sshd port). It generates a temporary file containing the stream of data read from every such channel read by the replay group. The tool performs channel-specific parsing: for instance, it decodes the X messages to read the corresponding key press events, and it intercepts data returned from functions such as `SSL_read` to retrieve the unencrypted data from ssh sockets. The channel input stream file therefore contains textual data that corresponds to the input.

Arnold follows a similar strategy to generate output stream files for a replay group. Understanding GUI output turned out to be tricky, however, because most programs we looked at did not send text to the X server, but instead sent binary glyphs generated by translating the output characters into a particular font. Arnold identifies these glyphs as they are passed to standard X and graphical library functions. It traces the lineage backward from these glyphs using one of the above linkages (e.g., the index linkage). These values are typically influenced by either or both of (a) textual input to the process being reexecuted, or (b) standard font files. By tracing glyphs to either location, Arnold recovers the characters associated with those glyphs. If the lineage is traced to a font file Arnold must determine the font character from the font file. This requires Arnold to understand font file formats (of which there are relatively few). Thus, it translates the sequence of glyph outputs to the underlying text they depict.

4.8 Query Execution

Arnold queries allow users to recall prior state and lineage information. There are three types of queries:

- **State queries** retrieve past transient state, persistent state, inputs, or outputs of the computer system.
- **Backward lineage queries** start at any current or past state, input, or output and trace the lineage of the bytes comprising that state backward in time according to a specified linkage function. A backward query answers the question: *How was this state derived?*
- **Forward lineage queries** start at a past state, input, or output and trace the lineage forward in time according to a specified linkage function to return current and past state and outputs derived from that state. A forward query answers the question: *What did this state influence and how was that influence propagated?*

4.8.1 State queries

Arnold can recover past file versions, transient process state, inputs, and output. If a specified file version does not already exist in its cache, Arnold uses cooperative replay to regenerate the contents of the file. Arnold reexecutes the specified replay group to regenerate both transient process state and output. Inputs (with the exception of file data) are logged, so no reexecution is needed to retrieve them.

Arnold also provides an interactive state query to allow users to inspect GUI output. During replay, X server output is displayed on the current screen, allowing the user to observe the display as it was manipulated during recording. Via a separate console, the user can fast forward to a particular output (i.e., the display is updated at replay speed without the original think time, I/O delays, etc.) or pause the replay at a particular point in the execution. By delaying a specified amount after each X output, Arnold can also display a slow-motion execution.

4.8.2 Backward lineage queries

A backward query starts from a specified piece of current or past state. The first step in processing this query is to translate the starting state into a set of <replay group,system call,byte range> tuples, where the system call is a specific call executed by a process in the replay group, and the byte offset is a range of bytes returned by that call. There are many possible types of starting state. For instance, the starting state may be data in a current file specified as a <filename,starting offset,length> tuple. Arnold looks up the lineage of these bytes in the current filemap, which contains the <replay group,system call,byte offset> tuple of the system call that produced each byte.

The starting state may also be process inputs (e.g., data read from a socket, pipe, or terminal). The user can specify a replay group and a specific type of input (e.g., network data) or a specific input channel (e.g., the terminal device), as well as a regular expression over the inputs of that type or sent over the channel. In this case, Arnold uses the user-interface tools described in Section 4.7 to return a set of <replay group,system call,byte range> tuples that match the specification.

The starting state may also be process outputs, which are specified in a similar fashion and translated to tuples via replay of the group and application of the user-interface tools described in Section 4.7. Past file versions can be used as starting state by either specifying the output or input of the file data.

For X output, the user may use the GUI replay described previously to regenerate past GUI output, pause the replay, and click on the display to specify an <x,y> coordinate. Arnold identifies the most recent X output generated by the group at an area surrounding the <x,y> coordinate. The starting state is the bytes passed to the system calls or library functions generating that output.

Finally, the starting state can be arbitrary process state in the replayed group. Custom state must be specified via a Pin tool. For instance, we describe a case study in Section 6.4.3 in which we determine what data was leaked by the Heartbleed vulnerability. The Pin tool inspects program state at the faulty instructions and determines whether any bytes were sent over the network incorrectly. If so, those bytes are specified as starting state for a backward lineage query to determine what specific data was leaked.

Given one or more of the above starting states, Arnold translates that state into a set of <replay group,system call,byte range> tuples. It executes a deterministic re-

play for each unique replay group in the set. If the user specifies a specific linkage function as part of the query, Arnold uses the Pin tool associated with that linkage. The linkage taints all inputs to that process group during replay by assigning a unique identifier for each <system call,byte> read from an input source. When an output matching one of the target tuples is generated, Arnold outputs the set of taint identifiers (if any) associated with the target bytes.

If the user declines to specify a linkage function, Arnold runs multiple replays, stating with the most restrictive linkage and proceeding to less restrictive ones as long as no target output is influenced by any input. Thus, if the copy linkage returns no values, the data linkage is run, then the index linkage, etc.

When inputs are found to influence target outputs, Arnold checks the source of those inputs. If the input is from an external source (e.g., bytes read from a network socket), lineage can be traced no further. If the input is from an IPC system call or file read, Arnold determines the replay groups that generated the data from the replay group log. It then recursively replays those groups to trace lineage back one more step. If the input is from a file, Arnold also reads from the log the system call and byte offsets that wrote the data. Otherwise, such information can be obtained by replaying the specified group.

Finally, if Arnold traces lineage back to a user input from the GUI or keyboard, it attempts to infer a linkage between the data entered by the user and any information recently seen by the user. It searches through the output of applications currently displaying output at the time the input was entered using the user-interface tool of Section 4.7. It performs a fuzzy string matching to determine whether the output likely influenced the input. If a match is found, it reports this as a *human linkage* and continues to trace the lineage back from the system calls that generated the matching output. Because human linkages are imperfect and may report both false positives and false negatives, Arnold treats human linkages as the weakest form of linkage, giving them the lowest priority with respect to matching inputs to outputs.

In summary, a backward query proceeds in a tree-like search, fanning out from one or more target states. The search continues until it is stopped by the user or all state has been traced back to external system inputs. As the search fans out, Arnold replays multiple replay groups in parallel. In addition, if no lineage is specified, it may test multiple linkages for the same group in parallel, terminating less restrictive searches if a more restrictive search finds a linkage.

4.8.3 Forward Queries

Forward queries start from some past state and return the values influenced by that state according to a specified linkage function. As with backward queries, the starting state may consists of current or past file state, process inputs or outputs, and/or specific bytes in an address space identified by a Pin tool.

Tracing intra-process lineage for forward queries is much more efficient than for backward queries because only those bytes corresponding to the target input need to be tainted. This results in much less overhead in tracking information flow within each process.

The lineage tool returns a set of outputs that are influenced by the target inputs. Arnold uses the reverse index described in Section 4.5 to determine which replay groups subsequently consumed that output. Thus, the query fans out recursively from the initial state to replay the groups that were influenced by that data. As with backward queries, human linkages from user-visible output to user-generated input are also traced as part of the query. Further, as the query fans out, multiple replay groups are reexecuted in parallel.

The forward query returns the set of all current state and system outputs influenced by the starting state. It also contains the specific chain (processes executed, inputs, and outputs) that propagated that influence.

5 Privacy

One concern with eidetic systems is the potential exposure of private data. Arnold's log can be used to recreate any state on the system and thus must be protected to the same or greater degree that one would protect other private memory and file system state.

A related concern is preserving a user's ability to exclude data from recording. For example, Arnold as described here would preserve all state from a user's "private" browsing session. However, Arnold could provide the ability to remove sensitive portions of the process graph by sanitizing and replacing portions of the log. For instance, Arnold could remove sensitive information contained in a process's address space by replacing that process's execution with a simple stub program that produces the same output.

A user who wants to remove sensitive information may actually benefit from Arnold's ability to track lineage. The user could identify the sensitive information or portions of execution, then ask Arnold to identify points in the process graph that depend upon this information. The user could then replace those parts of the graph with stubs that jump over the sensitive portions. Of course, once data is removed from Arnold, all lineage information and ability to query the execution are also lost.

6 Evaluation

Our evaluation answers the following questions:

User	Days	Groups per day	Storage utilization (MB) per day			
			RAW file cache	Logs	Filemap	Total
A	25	995	475	267	36	779
B	24	475	1095	936	339	2064
C	21	26122	869	350	690	1910
D	16	3339	1675	82	838	2594

Table 1: Storage utilization during multi-week trial

- What is Arnold's performance overhead?
- What are Arnold's storage requirements?
- What types of queries can Arnold answer?

6.1 Storage overhead

We first consider the storage overhead of running an eidetic system. To measure this overhead, the authors of the paper continuously recorded activity on their workstations for several weeks. The recorded activity included all user-level processes started from a terminal or launched from the GUI. It did not include several system-level processes, such as the X server (which is not recorded per the design in Section 4.2), processes that directly manipulate the replay data (e.g., indexing tools), and the sshd server (since we sometimes needed to log in without replay enabled for testing and maintenance). With the above exceptions, the vast majority of user activity was recorded. No files were evicted from the copy-on-RAW file cache since storage utilization was reasonable.

Table 1 summarizes the storage cost of our eidetic system. The third column shows the average number of replay groups created per day; note that there may be many threads and processes within a single group. The number of groups created varies widely depending on workload; some users have a few long-running applications; others have workflows (e.g., compilation) that create many short-lived groups. The next columns show average storage utilization per day, broken down to show the individual storage utilization of the copy-on-RAW file cache, the replay log storage, and the filemap. Process checkpoints are included in the total but not shown separately, since they account for less than 2 MB per day of storage. Executables and shared libraries referenced by checkpoints are included in the copy-on-RAW file cache.

While we cannot make comprehensive claims about storage utilization without a wider user study, this preliminary data is very encouraging. All users require less than 2.6 GB of storage per day; a 4 TB drive would suffice for 4–14 years.

6.2 Benefits of compression

Next, we quantify the benefits of Arnold's storage optimizations. Table 2 shows results for several workloads. The `Firefox` workload measures a half-hour browsing session consisting of 15 minutes of active browsing across 8 complex Web sites (e.g., Facebook and stack overflow), followed by 15 minutes of idle usage with Gmail windows open (triggering periodic JavaScript execution). The `evince`, `gedit`, `gpaint`, `LibreOffice` spreadsheet, and `LibreOffice` presentation workloads use those applications intensely for 3 minutes. The `latex` workload builds a prior OSDI paper, and the `make` workload builds the libelf-0.8.9 library. In these and subsequent experiments, we ensure repeatable results by automating all GUI workloads with the Linux Desktop Testing Project library [1], which captures and emulates GUI events.

The first row in the table shows storage usage when all nondeterministic inputs are logged without any compression. The subsequent rows show the effect of cumulatively applying model-based compression, copy-on-RAW file caching, the deterministic X proxy, semi-deterministic time, and gzip compression. The final row shows the compression ratio compared to baseline due to all of Arnold's optimizations.

Arnold averages a 411:1 compression ratio compared to the baseline. For comparison, simply applying gzip to the baseline averages only a 6:1 compression ratio. At 36:1, `LibreOffice` presentation sees the lowest compression ratio; this is due to the recording of temporary files written and read by the application. In the future, Arnold could omit such data since the files are never read by other replay groups and Arnold could regenerate the file contents during replay.

6.3 Performance overhead

Next, we measure Arnold's performance overhead during recording (i.e., the overhead that would normally be experienced by a user). All tests were run on a computer running Ubuntu 12.04LTS with an 8 core Xeon E5620 2.4 GHz processor, 6 GB memory, and a 1 TB 7200 RPM hard drive. We measured several terminal and GUI applications, and one server workload (apache):

- **kernel copy** – `cp -a` of the 3.5.0 Linux source.
- **cvs checkout** – check out Arnold's kernel source (589 MB, 52730 files) from a repository accessible via a local, 1 Gb local network connection.
- **make** – compile the libelf-0.8.9 library.
- **latex** – build a prior OSDI paper with `latex/bibtex`.
- **apache** – run the apache benchmark on an apache 2.2.22 server, configured with mpm_prefork with 256 workers, and a client connected via a 1 Gb local network connection (5000 requests for a 34 KB page with 50 concurrent requests at a time.)
- **gedit** – open a 15,000 line C file and find/replace on a commonly occurring string.

	Storage utilization (MB)							
	Firefox	evince	gedit	gpaint	spreadsheet	presentation	latex	make
Baseline	4517.63	194.51	764.34	95.29	4362.49	455.10	20.05	86.69
Model-based compression	308.93	7.07	12.50	36.12	41.16	31.07	7.19	81.52
Copy-on-RAW file cache	283.37	2.63	8.41	34.77	22.63	23.83	0.25	6.13
X compression	173.74	2.61	8.29	1.08	22.55	15.94	0.25	6.13
Semi-deterministic time	127.72	2.11	6.51	0.94	19.23	15.39	0.29	6.12
Gzip	24.87	0.11	0.50	0.08	3.33	12.71	0.05	1.15
Compression ratio	182:1	1752:1	1530:1	1217:1	1311:1	36:1	393:1	75:1

Table 2: Reduction in storage utilization via incremental application of optimizations

- **facebook** – load the White House's public Facebook page in `Firefox` version 23.0 (the completion time is measured by the onLoad event.)
- **spreadsheet** – Open a 704 KB csv spreadsheet in `LibreOffice` 3.5 and convert it to an xml document.

Figure 1 shows the performance of Arnold on a variety of desktop workloads, normalized to the performance of an uninstrumented system. The middle bar for each workload shows the performance during recording; this is the overhead the user will experience during normal operation. The third bar shows the performance during recording when Arnold uses a second hard drive for logging, which minimizes interference with normal file system writes.

Arnold's overall performance impact is small: overhead is under 12% with a single disk for all but two workloads. The cvs checkout has approximately 100% overhead with a single disk because it saves all checked-out data twice: once as nondeterministic network input and again when cvs writes the data to the file system. Adding a second logging disk reduces the overhead for cvs to 15%.

The higher overhead seen by kernel copy is caused by saving filemap entries. This workload is disk-bound and creates many small files. For each file created, Arnold must create a B-tree to record lineage data—this is effectively a worst-case for saving filemap entries. A separate logging disk reduces the overhead to 1.7%.

The Facebook tests contained some outliers due to external network and servers. We eliminated gross outliers (500% or more above the median) from our measurements (both for baseline and for Arnold); doing so did not help Arnold disproportionately.

We also evaluate the scalability of Arnold on several Splash 2 benchmarks, shown in Figure 2. While scalability was not a focus of our work, Arnold has low overhead for all these benchmarks up to 8 threads. We attribute this to two factors. First, Arnold requires programs to be race-free, so it only has to check and log inter-thread synchronization operations rather than all shared-memory operations. Second, Arnold's model-based compression reduces the instrumentation overhead per synchronization operation to only two atomic instructions in the common case.

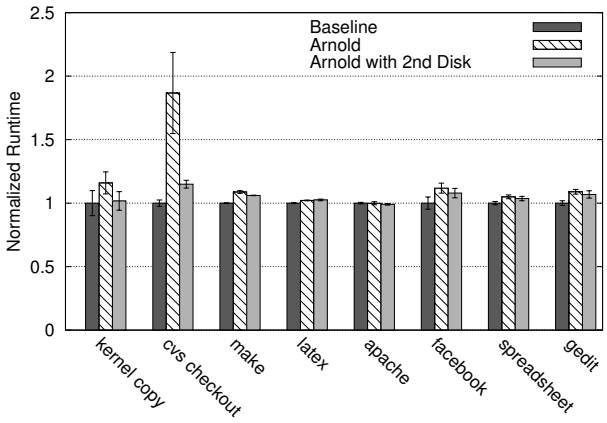

Figure 1: Arnold performance overhead normalized to unmodified Linux. Error bars are 95% confidence intervals.

In summary, Arnold adds modest overheads of less than 12% with a single disk on all but 2 workloads over a wide range of desktop and interactive applications. Adding a second hard drive reduces the overhead to under 8% on all but one workload. In practice, even on single hard drive configurations, we noticed virtually no difference between our recorded applications and non-recorded applications. In fact, we needed to add a utility to our shell interface simply to determine whether recording was currently enabled or disabled.

6.4 Case studies

Finally, we look at a series of case studies of queries that we expect to be typical of Arnold's usage.

6.4.1 Backward Query

Our first case study is a typical backward query. In this scenario, a colleague points out to the user that he has cited the wrong paper in a conference submission. The user runs a backward query to determine how the incorrect citation was produced and what the correct citation should be.

We executed this query by opening a binary document with `xdvi`, scrolling to the bibliography, and clicking on a screen location to specify the incorrect citation as the

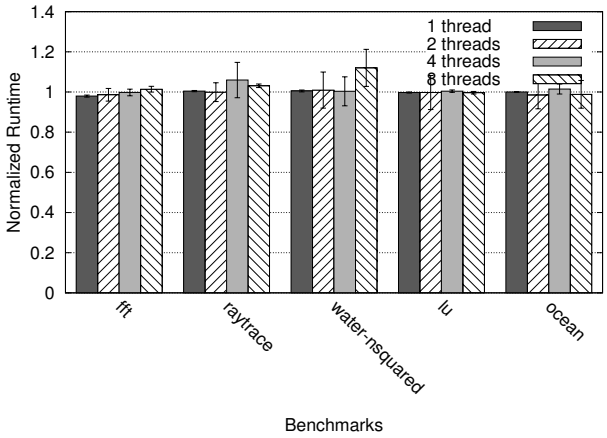

Figure 2: Arnold's scaling, normalized to unmodified Linux, on Splash2 benchmarks. Error bars are 95% confidence intervals.

starting state for the query. We did not specify a linkage, so Arnold ran the query with multiple linkage functions (the various linkage functions are analyzed in parallel via concurrent replay). For each step, Arnold chooses the most restrictive linkage function that produces some result (shown below in parentheses).

The query returned the following results:

- The specified output of `xdvi` came from the input file "paper.dvi" (index linkage).
- The incorrect citation in "paper.dvi" was generated by `latex` with data coming from the input file "paper.bbl" (data linkage).
- The data in "paper.bbl" was generated by `bibtex` with data from "full.bib" (copy linkage).
- The data in "full.bib" was generated by `vim` with data from the terminal device (copy linkage).
- A human linkage (described in section 4.7) reveals a fuzzy substring match between data coming from the terminal and `Firefox` output.
- The output displayed by `Firefox` came from a conference Web site (data linkage).
- The query also reports four false positives: a `latex` format file, a font file, `libc.so` and `libXt.so`.

Using Arnold, the user fast-forwards a `Firefox` replay to the point indicated by the query result. On viewing the recreated GUI, he realizes that the paper that he meant to cite was the *next* paper in the session after the incorrect citation.

As shown in the first row in Table 3, the query takes 209 seconds to execute, whereas the cumulative execution time of the recorded processes was only 96 seconds. Replay of the processes with zero instrumentation takes only 2 seconds because all user think time and most I/O delays are eliminated or replaced with a sequential disk read from the log. Simply attaching Pin to the replayed processes and inserting a very minimal instrumentation tool (which counts the number of system calls executed) increases the replay time to 70 seconds—this is the lower bound for any Pin tool on this workload.

The time it takes Arnold to perform the queries in table 3 is dominated by the Pin tool instrumentation and analysis, and not the actual replay system. Consequently, Arnold's query times are dictated by the number of instructions analyzed and the amount of taint information in the address space.

This query demonstrates that Arnold can successfully follow a long chain of applications to trace the lineage of data back to external inputs. The chain contains both binary and text data, as well as several types of linkages (intra-process, file, and human). Note that simply searching over inputs and outputs cannot reveal this whole chain (e.g., incoming Web data is encrypted, text input to `vim` includes backspaces and various keyboard macros, etc.) Lineage queries, however, can uncover linkages to such inputs because they directly observe the transformation of bytes in the process address space.

6.4.2 Forward Query

The second case study is a typical forward query. Our user now wishes to determine what other data and output has been affected by the faulty citation.

We executed this query by specifying the starting state as the incorrect citation in "full.bib." We also specified the index linkage function.

The forward query returns a list of external outputs and current files that the incorrect citation affected. Some key points of the result are:

- All subsequent versions of "full.bib" contain the incorrect citation. This is a shared bibliography file that is used to generate citations in several other papers on the user's computer. The forward query tracks the incorrect citation through the entire paper compilation process (e.g., though `bibtex`, `latex`, `dvips`, and `ps2pdf`).
- The query flags all files produced during the paper compilation process that include the specified citation (e.g., "paper.bbl", "paper.dvi", "paper.ps", and "paper.pdf.")
- The query does not return false positives. The user also has several papers that use the bibliography file "full.bib", but those papers do not cite the incorrect citation. Even though "full.bib" is read when those papers are compiled, Arnold correctly reports that no output file is affected by the incorrect citation.
- The query shows that the user had copied and pasted the incorrect citation from "full.bib" to another file, "paper.bib", using `vim`. The query also returns subsequent compilations and output files of

Case Study	Record Time	Replay Time	Replay & Pin	Query Time
Case Study 1: Backward Query	96.1s	2.2s	70.0s	209.5s
Case Study 2: Forward Query	30.3s	1.6s	80.4s	110.7s
Case Study 3: Forward Heartbleed Query	114.1s	0.1s	6.9s	19.7s
Case Study 3: Backward Heartbleed Query	230.3s	0.4s	139.5s	235.1s

Table 3: Summary of case studies

those compilations that reference the incorrect citation in "paper.bib".

- The query detects that the user ran a python script to produce a file with more succinct version of the citations, "small.bib", from "full.bib". It detects the incorrect citation in "small.bib" and in paper compilations that reference the incorrect citation from that file.
- The query detects that the user e-mailed a paper with the incorrect citation. This shows up as a network output of `sendmail`.
- The query returned no false positives.

The second row of Table 3 shows that the forward query required 111 seconds to execute, whereas simply replaying all processes with the simple Pin tool require 80 seconds. Thus, the relative overhead of the forward query instrumentation, is (as expected) much less than that for a backward query.

6.4.3 Heartbleed

Our third case study is motivated by the 2014 Heartbleed attack. One reason this attack caused such concern is that service providers were unable to determine what (if any) data was leaked. We show how Arnold is able to help an administrator determine whether sensitive data was leaked from a low-volume Web server, which hosts and stores a key-value database.

First, the administrator runs a forward query to see if the server's private key was leaked. This query requires a custom definition of output, so she creates a Pin tool. Heartbleed exploited a missing bounds check, so the Pin tool simply emulates the missing bounds check when the target instruction is reached and flags as output any data in excess of what the bounds check would have rejected. Her forward query specifies a starting state of the private key file, an output definition of only those bytes returned by the Pin tool, and the index linkage function.

We emulated this scenario by recording 100 GET and POST requests to Nginx 1.4.7 with OpenSSL version 1.0.1f (run times scale roughly linearly with the number of requests). This query took 20 seconds to perform and returned no outputs, showing that the private key was not leaked). We confirmed the correctness of this result manually.

Next, the administrator asks: *was any data leaked, and if so what data?* We constructed a backward query to answer that question. We used the custom Pin tool to define as starting state any data incorrectly sent due to the Heartbleed exploit and specified the index linkage. The backward query determined that:

- The Web server, `Nginx`, serviced a heartbeat request that leaked process memory.
- The leaked bytes came from a UNIX socket written by `FastCGI`, which is responsible for dynamic Web content.
- `FastCGI` received these bytes from a pipe written by a python script that it spawned.
- The script read the bytes from a database file.
- The bytes read from the database file came from POST requests that inserted those key-value pairs. This is determined by following the bytes back through a python script, `FastCGI`, and `Nginx`.

In summary, Arnold reveals both the leaked content *and* the origin of that data. Further queries could reveal the specific users who had their content leaked (e.g., by using a Pin tool to extract the userid from the connections that wrote the leaked data). The total query time was 235 seconds, roughly double the cost of replay and Pin instrumentation alone.

This case study shows the value of not limiting a priori the types of lineage that an eidetic system can track. For example, prior tools for intrusion recovery focus on inter-process lineage but cannot track intra-process lineage [23, 18, 22]. Upon learning of the vulnerability, the user can write new tools that detect data flows she had not anticipated at the time the system was being recorded, then use these tools on executions that were recorded before the tools existed.

7 Conclusions and future work

Arnold is a prototype of an eidetic system, targeted at personal computers and workstations. Arnold can recall any past user-level state, and it can trace the lineage of any byte in a current or prior state. This paper shows that the overheads of providing such functionality are reasonable: our results shows that adding a commodity hard drive can satisfy 4 or more years of storage needs with most runtime overheads under 8%. We have made the source code for Arnold available at https://github.com/endplay/omniplay.

Our case studies show the power of an eidetic system by recovering past state and tracing the lineage of data through a wide variety of applications and user interactions. The precision of combining operating system tracing of inter-process information flow with retrospective analysis of intra-process information flow yields accurate and informative query results.

We plan to explore several new aspects of eidetic systems. First, while Arnold can track all computation on a single machine, it cannot connect computation between multiple machines. One direction for future work is how to efficiently achieve Arnold's lineage tracking within large distributed systems. A second direction of future work is how to enable users to retroactively remove data from Arnold's log in order to preserve privacy. A complementary direction of future work is how to prevent or detect tampering with Arnold's log [20].

8 Acknowledgments

We thank the reviewers and our shepherd, Lorenzo Alvisi, for their thoughtful comments. This material is based upon work supported by the National Science Foundation under grants number CNS-0905149 and CNS-1017148. Yihe Huang created data race tools that we adapted for this work. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Linux Desktop Testing Project. `http://ldtp.freedesktop.org`.

[2] ADAMS, I. F., LONG, D. D. E., MILLER, E. L., PASUPATHY, S., AND STORER, M. W. Maximizing efficiency by trading storage for computation. In *Proceedings of the Workshop on Hot Topics in Cloud Computing* (June 2009).

[3] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 193–206.

[4] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation* (Hollywood, CA, October 2012).

[5] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[6] BACON, D. F., AND GOLDSTEIN, S. C. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging* (1991), ACM Press, pp. 194–206.

[7] BENT, J., THAIN, D., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIVNY, M. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation* (March 2004).

[8] BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications* (Orlando, FL, October 2009), pp. 81–96.

[9] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel Java. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications* (Orlando, FL, October 2009), pp. 97–116.

[10] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 80–107.

[11] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008), pp. 1–14.

[12] CUI, H., WU, J., GALLAGHER, J., GUO, H., AND YANG, J. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 337–351.

[13] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009), pp. 85–96.

[14] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[15] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M., AND CHEN, P. M. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2008), pp. 121–130.

[16] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[17] FELDMAN, S. I., AND BROWN, C. B. IGOR: A system for program debugging via reversible execution. In

PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (1988), pp. 112–123.

[18] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The Taser intrusion recovery system. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005).

[19] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[20] HAEBERLEN, A., ADITYA, P., RODRIGUES, R., AND DRUSCHEL, P. Accountable virtual machines. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[21] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th International Symposium on Computer Architecture* (June 2008), pp. 265–276.

[22] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[23] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 223–236.

[24] LAADAN, O., BARATTO, R., PHUNG, D., POTTER, S., AND NIEH, J. DejaView: A personal virtual computer recorder. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, Oct 2007), pp. 279–292.

[25] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)* (June 2010), pp. 155–166.

[26] LEE, D., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Chimera: Hybrid program analysis for determinism. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation* (Beijing, China, June 2012).

[27] LEFEBVRE, G., CULLY, B., HEAD, C., SPEAR, M., HUTCHINSON, N., FEELEY, M., AND WARFIELD, A. Execution Mining. In *Proceedings of the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2012).

[28] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 327–336.

[29] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[30] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture* (June 2008), pp. 289–300.

[31] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-based versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, CA, February 2009).

[32] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.

[33] NARAYANASAMY, S., PEREIRA, C., AND CALDER, B. Recording shared memory dependencies using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 229–240.

[34] NETZER, R. H. B. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (1993), pp. 1–11.

[35] NEWSOME, J., AND SONG, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium* (February 2005).

[36] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, March 2008), pp. 308–318.

[37] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009), pp. 97–108.

[38] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 177–191.

[39] PETERSON, Z. N. J., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transacations on Storage 1*, 2 (2005), 190–212.

[40] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the 1995 Winter USENIX Conference* (January 1995), pp. 213–223.

[41] POZNIANSKY, E., AND SCHUSTER, A. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPOPP03* (San Diego, CA, June 2003), pp. 179–190.

[42] RONSSE, M., AND DE BOSSCHERE, K. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems 17*, 2 (May 1999), 133–152.

[43] RUSSINOVICH, M., AND COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (1996), pp. 258–266.

[44] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. *SIGOPS Operating Systems Review 33*, 5 (1999), 110–123.

[45] SEREBRYANY, K., AND ISKHODZHANOV, T. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (December 2009).

[46] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND ZHOU, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 29–44.

[47] VAHDAT, A., AND ANDERSON, T. Transparent result caching. In *Proceedings of the 1998 USENIX Annual Technical Conference* (June 1998).

[48] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, October 2011).

[49] VEERARAGHAVAN, K., FLINN, J., NIGHTINGALE, E. B., AND NOBLE, B. quFiles: The right file at the right time. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (San Jose, CA, February 2010), pp. 1–14.

[50] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

[51] VLACHOS, E., GOODSTEIN, M. L., KOZUCH, M. A., CHEN, S., FALSAFI, B., GIBBONS, P. B., AND MOWRY, T. C. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, March 2010), pp. 271–284.

[52] WEERATUNGE, D., ZHANG, X., AND JAGANNATHAN, S. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2010), pp. 155–166.

[53] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th International Symposium on Computer Architecture* (June 2003), pp. 122–135.

[54] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *MOBS07* (June 2007).

[55] ZAMFIR, C., AND CANDEA, G. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th ACM European Conference on Computer Systems* (April 2010), pp. 321–334.