Automated Assessment of Secure Search Systems

Mayank Varia
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
mayank.varia@ll.mit.edu

Benjamin Price
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
ben.price@ll.mit.edu

Nicholas Hwang
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420
nicholas.hwang@ll.mit.edu

Ariel Hamlin
MIT Lincoln Laboratory
ariel.hamlin@ll.mit.edu

Jonathan Herzog*
Jonathan Herzog Consulting
jherzog@
jonathanherzog.com

Jill Poland
MIT Lincoln Laboratory
jill.poland@ll.mit.edu

Michael Reschly
michael@reschly.com

Sophia Yakoubov
MIT Lincoln Laboratory
sophia.yakoubov@ll.mit.edu

Robert K. Cunningham
MIT Lincoln Laboratory
rkc@ll.mit.edu

ABSTRACT
This work presents the results of a three-year project that assessed nine different privacy-preserving data search systems. We detail the design of a software assessment framework that focuses on low system footprint, repeatability, and reusability. A unique achievement of this project was the automation and integration of the entire test process, from the production and execution of tests to the generation of human-readable evaluation reports. We synthesize our experiences into a set of simple mantras that we recommend following in the design of any assessment framework.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors, testing tools*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms
Reusable libraries, Empirical software validation, Software design tradeoffs, Information search and retrieval

1. INTRODUCTION
Cyber security system evaluation is a recognized hard problem that, if solved, will enable the "field to gauge its progress towards handling security threats" [4, 7]. Advances are needed in reducing the time to evaluate systems, especially research prototypes with rapidly evolving requirements and designs. Today's testing requires extensive human effort to develop software to run tests, analyze test results, and write reports and presentations.

In this paper, we describe advances developed to enable rapid and repeatable test and evaluation (T&E) of research-quality privacy-preserving data search systems. Our contributions include the design for an assessment framework that automatically generates test case inputs, exercises prototypes within an instrumented environment, compares prototype performance relative to a baseline technology, and generates human-readable evaluation reports within 1 hour of test completion.

Additionally, we synthesize our experiences into a set of software T&E mantras that we believe all software assessment frameworks should uphold. We believe that our recommendations will help others develop similar assessment frameworks more accurately and rapidly in the future.

2. SOFTWARE T&E MANTRAS
We first present our software T&E mantras. These are *fundamental principles* that we believe should be incorporated into all software T&E processes, tools, and frameworks.

End-to-end automation. Build integrated "turnkey" interfaces with which an operator can define test parameters, push a button, and watch the test complete.

Repeatability. Tests should be fully repeatable. Test data, cases, and environments should be fully recoverable and quickly replicable.

Black box treatment of systems. Do not depend on the low-level operation of any system under test. Instead, expose and document an interface that systems use to interact with the test framework.

Test-time agility. Time during the test period is precious. When planning tasks for a T&E program, spend a week of work to save a few hours of test time.

Real-time status reports. Operators should easily be able to (1) determine the status and progress of each test case and (2) inspect the state of each system under test and its test environment.

*Work performed while at MIT Lincoln Laboratory.

This work is sponsored by the Intelligence Advanced Research Projects Activity under Air Force Contract FA8721-05-C-002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Copyright is held by the authors.

Minimal overhead. Ensure that the test software does not impact the system under test or influence any performance metrics being captured.

Extensibility. To accommodate new test requirements, enable rapid addition of new test parameters, test cases, and environment configurations.

Rapid environment reconfiguration. Build a rapidly deployable plug-and-play framework wherein an operator can quickly change performer systems, test parameters, or test environments.

Smoke testing. Prepare tests that can quickly prove the stability of the system under test and its interoperability with the assessment framework. This can identify bugs and reduce risk early in the test.

Comprehensive data capture. All test inputs, environment configurations, events, and results should be fully captured and methodically archived to facilitate robust post-test analysis. Test artifacts should be traceable to a particular time, set of test inputs, and environment configuration.

Reusability. To the extent possible, build a framework that enables reuse by future T&E work.

The rest of this paper presents our T&E experience in assessing privacy-preserving database systems. We encourage readers to view this exposition, especially the evolution of our assessment framework over time, as a case study that motivates, supports, and justifies the mantras described above.

The description of our T&E experience is organized as follows: Section 3 gives an overview of the SPAR program for privacy-preserving data searches, Section 4 describes the methodology used and lessons learned during our T&E effort on SPAR, and Section 5 details our assessment framework components.

3. SPAR PROGRAM OVERVIEW

In this section, we describe the project for which we performed our T&E work, in order to provide context for our assessment and explain the driving factors for our framework. The Security and Privacy Assurance Research (SPAR) program was launched in 2010 by the Intelligence Advanced Research Projects Activity. The program objective for SPAR was to design and build new privacy-preserving data search technologies that are fast and expressive enough to use in practice. The program comprised nine research teams who worked on three Technical Areas (TAs). In this paper, we focus on TA1, the component of SPAR that designed privacy-preserving database systems.[1]

TA1 performers built database management systems that enabled clients to issue SQL-style queries to a database server (and a third party intermediary data store) and retrieve all records matching its query. The performers' software was required to operate within a 10× performance factor of a non-privacy-preserving system on a 10 TB database, while also providing several security benefits [6]:

- The client must learn almost nothing about records that do not match its query.
- The server must validate that the client's query matches a policy without learning anything else about the query or the records returned.
- The third party must learn almost nothing about either the contents of the database or the queries.

In order to meet these objectives, TA1 performers used novel cryptographic techniques that enabled queries to be executed directly on an *encrypted* copy of the database held by a third party [2, 9]. A variety of query types were supported, including equality, range, keyword, stem, string wildcards and substrings, ranking, and XML queries, as well as boolean combinations of the above types (`and`, `or`, and threshold). All queries were supported for `select *` and `select id` requests, where the `id` field was guaranteed to have a unique value for each record. Lastly, performers supported database modifications, including row insertions, updates, and deletions.

4. SPAR TEST & EVALUATION

We at MIT Lincoln Laboratory did not serve as a performer who built a privacy-preserving search tool as described above; instead, we participated as the T&E team for all nine SPAR performers. Our evaluations for each performer comprised two parts: (1) a review of their cryptographic algorithm's security guarantees, and (2) an empirical assessment of their software's correctness, functionality, and performance. This paper focuses exclusively on the latter aspect of our evaluation.

SPAR was split into two "phases" (Phases 1 and 2) lasting about 1.5 years each, and Lincoln conducted empirical evaluations [7] of every performer's software at the end of each phase. To do so, we built an *assessment framework* that measured each performer's software and reported whether it met the SPAR program's requirements [6].

In this section, we discuss (1) our overall test execution process, (2) our Phase 1 assessment framework design, (3) the lessons learned from our Phase 1 experience, and (4) our improved Phase 2 assessment framework. We defer detailed descriptions of our assessment framework components to Section 5.

4.1 Test Execution

In this section, we discuss how we performed our tests. One week each was allocated per performer system for testing, with a small representative team from performers onsite for the test. Each test was run with the performers' software on Lincoln hardware. Our assessment framework then connected to each machine (client, server, and third party) to control the execution of tests and archive all results.

Prior to the testing period, we prepared synthetic databases and queries over that data to use during testing. Additionally, we partitioned the queries into *test scripts* based on the genre of query executed (keyword, boolean, range, etc.) and the expected number of results for each query. In order to

[1] Most of Lincoln's assessment framework design principles & lessons learned apply to the other two TAs as well [6, 13].

properly characterize performers' database systems, our assessment framework executed each test script twice, first in a "latency" mode in which performers were given one query at a time, and second in a "throughput" mode in which performers were given queries as quickly as possible to measure their system's parallelization.

At the beginning of each test week, our first action was to run a *smoke test* made up of scripts containing a small set of queries that represented all the query types that would be executed during the full test. Due to the prototype nature of these systems, we found that performers usually made several patches during the test week to (1) communicate properly with the assessment framework and (2) fix correctness bugs. By exposing problems early in the test week, the smoke test gave performers adequate time to develop patches. Hence, smoke testing, and subsequent debugging, often consumed the first day or two of testing.

During formal testing, we maintained a set of logs to capture test events and manage the chaos of the test week. Performer systems can fail in unpredictable ways, and *comprehensively capturing* why and when anomalies occurred was vital for root-cause analysis and planning future test days (for example, to determine which tests to re-run after a performer deployed a patch). Additionally, logs helped determine which test results should be discarded, perhaps due to performer bugs that were later patched or because a test was executed improperly.

The post-test process of tracing events to logs and filtering out invalid test results for analysis proved to be complex and time consuming, particularly in Phase 1 when the level of automation in our framework was lower. Therefore, in Phase 2 we were determined to increase the *end-to-end automation* of our testing procedure, particularly to improve our team's *test-time agility* and maximize human time available during the turbulent and fast-paced testing process.

4.2 Phase 1 Framework

We developed the Phase 1 assessment framework for TA1 based upon three primary guidelines. First, we wanted to *capture comprehensive* measurements for query latency and throughput at the 10 TB scale with *minimal overhead*. Second, we needed to consider the unique difficulties of testing privacy-preserving systems; for example, encrypted communications between parties precluded the test framework from performing traffic inspection, and necessitated *black box treatment of systems*. Third, recognizing the need to expand our framework in Phase 2, we planned for *extensibility* in our design to make additions as seamless as possible.

Our Phase 1 test framework, shown in Figure 1, had three central components: a data generator, a baseline comparison system, and a test harness.

First, our *data generator* built artificial but realistic-looking data corresponding to people's names, addresses, birthdays, etc. based upon data from the US Census Bureau [10, 11] and Project Gutenberg [8] in a *repeatable* fashion [12].

Then, we built a MySQL database system as a *baseline* for comparison. Our decision of MySQL was motivated by a desire to use open-source software (for ease of distribution) and to simulate an "off-the-shelf" solution that one might deploy with little customization if privacy were not critical.

Next, our *test harness* actuated and monitored the performer and baseline systems. It used the previously generated test scripts to coordinate the execution of queries and database updates. Detailed, timestamped outputs would be saved in *test logs*. We optimized the test harness' performance so that it could handle massive string I/O with minimal CPU, RAM, and disk usage.

We had three additional components that completed the framework design. First, our *query generator* created queries that were well-distributed among several variables of interest, such as the number of records returned. Second, our *log parsers* transformed test logs into comma-separated value files containing raw results and performance data. Third, our *data visualization tools* produced graphs and tables summarizing each system's performance relative to the baseline. For more details on the Phase 1 framework components, see Section 5.

4.3 Phase 1 Lessons Learned

Our Phase 1 framework upheld only some of the mantras in Section 2 of this paper. As a result, it had some inefficiencies that we could only overcome with tedious human effort before, during, and after testing.

First, query generation and test script preparation were manual, error-prone processes (i.e., lacked *end-to-end automation*). Queries were generated without prior knowledge of the responses and had to be vetted against the baseline to determine the correct responses. Another related issue was that we had little idea of how long test scripts would take to run if they were not previously executed against the baseline. As a result, we sometimes were forced to "test blind" without ground truth knowledge, and sometimes struggled to accommodate last-minute test changes.

Second, during test execution, simple shell scripts were written to deploy system components on various machines and run each test case. However, these scripts quickly became unwieldy because each performer had different system requirements that were prone to change many times during a test (e.g., package dependencies, prerequisite files, network setup). Other dynamics such as the need to re-run a very specific selection of tests, point to newly patched performer software, change output directories, and switch in and out of "debug" mode necessitated continual modification of these scripts (i.e., we lacked *rapid environment reconfiguration*). As a result, configuration management was difficult, and our test artifact archives quickly became disorganized.

Third, due to insufficient logging of the *ad hoc* decisions made during testing (see Section 4.1), post-test analysis was burdensome (i.e., lacked *comprehensive data capture*). Weeks were spent aggregating test artifacts and determining which should be discarded due to human error or performer bugs. Additionally, the process of using our data visualization tools to produce an easily-readable report was extremely human-time intensive, and we needed to iterate over this process many times because deficiencies in our data

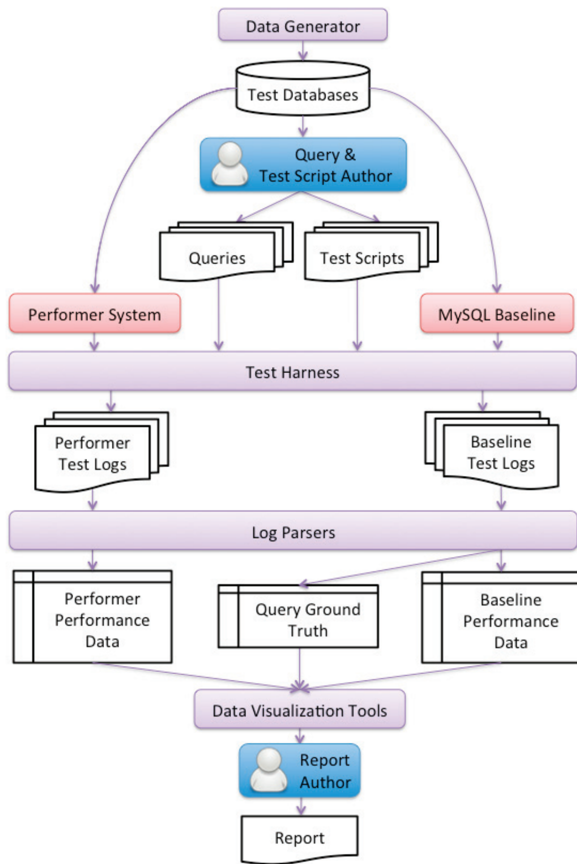

Figure 1: Phase 1 test framework (see Section 4.2)

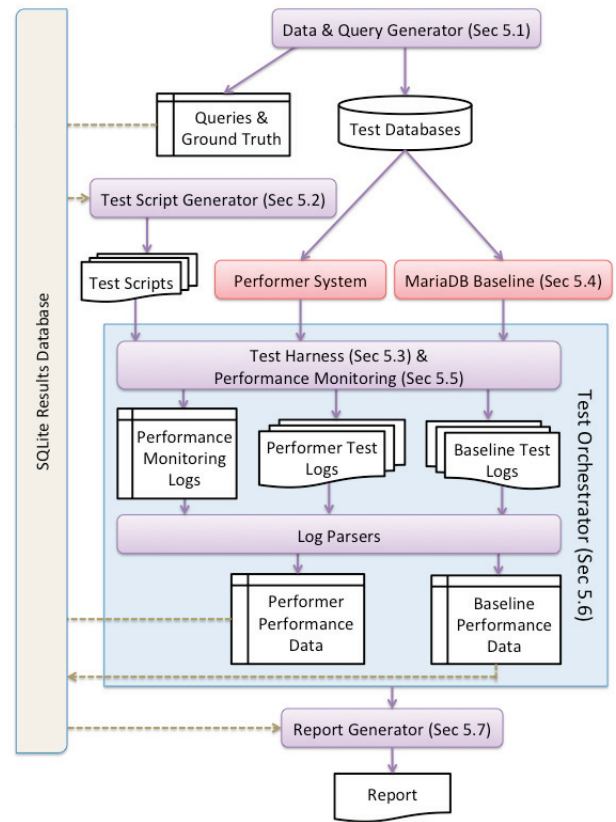

Figure 2: Phase 2 test framework (see Section 4.4)

selection would often only be found from a readable report.

4.4 Phase 2 Framework

Based on the lessons learned in Phase 1, we imposed two new primary guidelines for our Phase 2 improvements: (1) automating each component to minimize the amount of human interaction required, and (2) integrating components to enable a single end-to-end test process.

We improved the *automation* of our assessment framework in four ways. First, we built a query generator that could automatically produce queries for each test database based on a single high-level description of the desired query types. Second, we automated and standardized the process of test script generation so that each test would run for an average of 30 minutes. Third, we built an automated test orchestrator that deployed and monitored test systems based on a performer-provided specification of their system's requirements. Fourth, we built a turnkey report generator that automatically produced a sponsor-ready LaTeX report.

We integrated the framework components together through a lightweight, portable SQLite database that served as a central repository for information between all framework components. Each component stored its results in the SQLite database for use by other components.

With these improvements, Phase 2 tests were conducted with great success. The *end-to-end automation* increased our *test-time agility* and enabled us to observe *real-time status* during tests, while the *comprehensive data capture* allowed us to deliver a report to the sponsor within minutes of the end of a test.

5. T&E SOFTWARE COMPONENTS

In this section, we describe the detailed design of each component of our software assessment framework. We also explain the transformation of each component throughout the two phases of the program for improved automation and integration.

Many components were designed with *reusability* in mind, and readers are encouraged to explore our assessment framework's open source repository at github.com/mitll-csa/sparta. We believe that many components, especially the synthetic data generator and knots data structure, can be used in other assessment efforts to better uphold our recommended software T&E mantras.

5.1 Data & Query Generator

In order to assess performer systems, we needed to construct three corpora of data: (1) raw data that systems could ingest as "rows" into a relational database, (2) a set of queries over those rows, and (3) "ground truth" answers and metadata for each query. This section describes the requirements for these corpora and the tools we built to generate them. More details can be found in a separate paper [5].

Requirements. The set of tests we were interested in required a variety of database sizes ranging from 10^3 rows to 10^9 rows. Each database had to be generated in an agreed upon schema and file format that could be easily ingested by performer systems and the MySQL baseline. Lastly, each database needed a set of "extra" rows that could be used as new data for the database modification tests.

Additionally, we needed to generate a set of queries to test each database. Test queries needed to be sufficiently varied in query type and result set sizes. Additional complexity was required for compound queries, as each individual clause in the compound query had similar diversity requirements independent of the overall query's requirements. For instance, we might desire conjunction queries of the form "$A \wedge B$" where $|A| \approx 100$, $|B| \approx 1000$, and $|A \wedge B| \approx 10$. To complicate matters further, each performer imposed different restrictions on their support for each query type.

The project's schedule and end-goals imposed another set of requirements. Rows and queries needed to be "realistic" (i.e., illustrate an actual use case with privacy concerns) so that our assessment would be more relevant to potential end users of the technology. Furthermore, the exact database contents and test queries needed to be unpredictable so that performers could not "game" the evaluation. The project's schedule requirements further dictated that the generator be built in a language that would enable rapid development (we chose Python) and be optimized and parallelized.

Data generator. We searched for existing datasets or automatic data generators, but none operated at the scale we needed. Hence, we built a new data generator that would produce a set of realistic but *synthetic* rows. To ensure data realism, our generator contained "learner" modules that trained on data sets provided by the US Census Bureau [10, 11] and public-domain novels from Project Gutenberg [8]. Learners generated probability distributions for each field of interest, and a "row generator" used these distributions to generate rows. These rows were submitted to a map-reduce framework that executed jobs (which we called "aggregators") on each row. One of these aggregators was responsible for writing the rows out to files that were eventually used to create test databases. For *repeatability*, data generation was parameterized by a random seed; this allowed performers to generate identical databases to ours and thus obviated the need to transmit large data sets.

Performance-wise, our Phase 1 data generator exploited parallelism to build 10 TB of synthetic data in about 1 day on a 12-core machine. By profiling our data generator to determine its bottlenecks, re-writing a small percentage of critical code in Cython (cython.org), and performing other optimizations, we were able to reduce the runtime of the generator to 9 hours in Phase 2.

The data generator was built in an *extensible* fashion. By gathering new training data and implementing new learners, it can be *reused* to generate a wide variety of synthetic test data sets such as realistic network traffic, HTML files, and file system contents. Also, our map-reduce framework [3] permits aggregation of custom statistics on the generated data, a feature that we utilized in Phase 2 query generation.

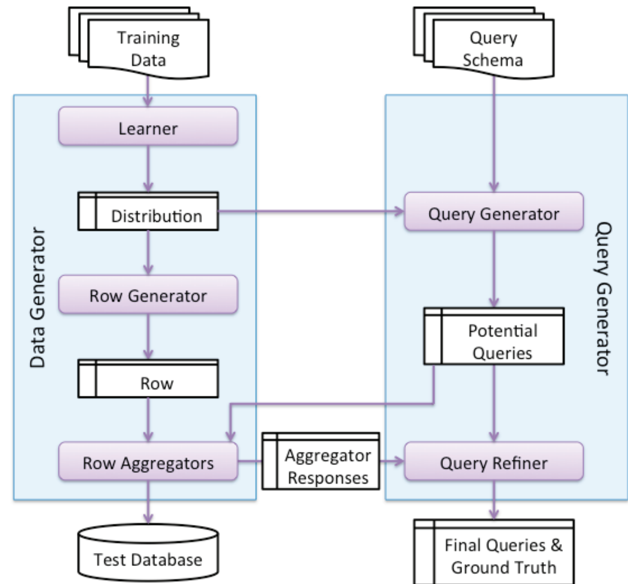

Figure 3: Shows the data interactions between the data generator and query generator.

Query generator. Our Phase 1 query and ground truth generator required three steps. First, we ran the data generator described above. Second, we inspected the generated database to find queries that fit our desired needs for result set size. Third, we ran these queries against our baseline system to determine the rest of the ground truth information, such as the contents of records that match the queries.

This procedure was insufficiently *integrated* and *automated*. The act of inspecting an already-generated database was wasteful: it took a few hours to run and did not utilize our ability to inspect database rows as we were generating them. Additionally, to collect sufficiently many queries, our inspection procedure had to be run several times in a manual fashion that took many weeks of human and CPU effort.

In Phase 2, we built a three-stage query generation system on top of the Phase 2 data generator, shown in Figure 3. First, it generated potential queries matching a simple user-specified "query schema" that provided a concise summary of the types and numbers of queries to generate. These queries were created based on the distributions from which database rows were generated, rather than the Phase 1 approach of using statistics gathered on already-generated databases.

The second step of query generation was *integrated* with the data generator's new map-reduce framework [3]. It built highly *extensible* map-reduce jobs (which we called aggregators) that examined rows as they were being generated to aggregate desired information about each potential query. We used this capability to determine the exact ground truth of all potential queries, plus additional information like the number of records matching each clause of a compound query. This new query generator added only a few minutes of overhead to the data generation proceess for simple queries. While compound query generation took more CPU time, its *automation* enabled us to be more *agile* and spend precious human effort near the testing period on other tasks.

Finally, after data generation was complete, a post-processing "refinement" stage filtered out queries that did not satisfy the query schema's requirements. The remaining queries, along with their metadata (such as the number of clauses) and their ground truth (IDs and hashes for all rows the query should return) were used to initialize the SQLite integration database, which in turn served as an input to the test script generator.

5.2 Test Script Generator

With the above datasets prepared, we next built test scripts that controlled the execution order of queries and database modifications. In Phase 1, this was a manual process, which led to several issues. Scripts were prone to copy-paste errors, and the time required to construct and modify these scripts made us inflexible to last-minute test changes.

In Phase 2, we built a test script generator that automatically produced several types of test scripts that the test harness could run: lists of queries (to run in latency or throughput mode), and database modifications (inserts, updates, and deletes). To build query test scripts, the generator leveraged data that the query generator inserted into the SQLite integration database: query types, the number of records queries returned, and which queries performers supported on a particular database size. With this information, the generator grouped similar queries (e.g., by query type, approximate number of results per query, etc.) into test script files. The size of these groups was determined by query latencies gathered during a "risk reduction" exercise prior to the Phase 2 test. We built test scripts that would execute in approximately 30 minutes, with the intention that a minimal amount of test time and data would be lost if a test failed or crashed. Database modification test scripts were generated similarly from extra rows created by the data generator.

Additionally, the generator created a logical directory and file nomenclature for the scripts. This not only kept test data more organized, but also facilitated better integration with our log parsers and report generation tools.

5.3 Test Harness

To actuate the performers' software in Phase 1, we built a test harness that could spawn performer components, execute the aforementioned test scripts, and log timestamped results to a file. To facilitate *black box system treatment*, we only interacted with performer components through named pipe connections. We provided a thoroughly documented protocol by which (1) our test harness would send each component commands, and (2) each component would send results to the test harness. The test harness design did not change appreciably in Phase 2.

We decided to communicate with performer software via ASCII strings over named pipes for two reasons. First, SQL commands and query results are ASCII-based, and therefore made ASCII a natural data format to base our protocol upon. Second, we wanted to have a single common standard for all of the performers, both for *black box consistency* and for better *repeatability*.

However, this also lead to the biggest challenge we faced with the test harness, which was the massive amount of string I/O that would occur during thousands of query commands on databases as large as 10 TB. Our test harness had to efficiently concatenate query results with no *a priori* knowledge of the number or size of results. Moreover, it needed to send and receive messages with *minimal overhead* and latency so as to not influence the performer systems' performance.

Operation	String	Rope	Knot
Index	$O(1)$	$O(\log n)$	$O(\log n)$
Concatenate	$O(n)$	$O(n)$	$O(1)$
Delete	$O(n)$	$O(\log n)$	$O(1)$
Substring	$O(n)$	$O(\log n)$	$O(1)$[a]
Iteration	$O(n)$	$O(n)$	$O(n)$
Comparison	$O(n)$	$O(n)$[b]	$O(n)$

Table 1: Asymptotic comparison of operations over strings, ropes, and knots.

[a] If done at front or end of knot, else $O(\log n)$.
[b] Adds 5-10× overhead and does not fail fast.

Standard string representations, like `C/C++ char*`s or `STL` strings, were not well suited to this type of processing because string concatenation, substring extraction, and string erasure for these representations require a number of heap operations that is $O(n)$ with respect to the length of the string. We first explored using ropes [1], which could do most of these operations in $O(\log n)$ time. Unfortunately, ropes incur a larger memory overhead, and cannot perform "fail fast" comparisons (whereby a comparison could return `false` without traversing the entire string), which were especially important for us. In addition to those drawbacks, ropes further impose approximately a 5-10× overhead for string comparisons[2]. Hence, despite their overall runtime improvement, ropes proved insufficient for our needs.

Our solution was to build a new data structure that we affectionately named *knots*[3] because they were ropes with a "twist." The core structure of a knot is a mutable double-ended queue of nodes, where each node contains a reference counted pointer to a "strand" (an immutable substring). Knots offer several benefits over ropes and reduce many of the operations that were important to us to constant time (see Table 1 for runtime comparisons to other string representations). Additionally, fail fast string comparisons remained just as efficient as they were with standard strings. Our development of knots allowed us to communicate with performers' systems without adding significant overhead that may have masked system performance characteristics.

As text protocol processing always requires efficient string handling, our knots data structure offers potential *reusability* in any application that requires keyword matching amongst large and frequent buffer concatenations.

5.4 Baseline System

The baseline system was used to compare performer systems to a non-privacy preserving system. It supported the same

[2] www.sgi.com/tech/stl/Rope.html
[3] The authors have written a white paper detailing our implementation of knots, which can be provided upon request.

query types as the performers, and was designed to emulate a reasonably efficient off-the-shelf database management system.

In Phase 1, we used MySQL as the server component of the baseline system; in Phase 2, we switched to MariaDB due to its support for non-blocking queries.

We designed a custom client component that translated queries and their responses between the formats expected by the commercial server software and our test harness. While MariaDB natively supported many of the query types required for SPAR, we needed to enhance existing MariaDB functionality or build completely new functionality to support a few query types: threshold queries (aka M-of-N queries) where results must match at least M of N specified search terms; ranking queries in which the results of threshold queries are sorted by the number of matching search terms; and several types of queries over large text fields such as free keyword search, keyword search based on word stems, word proximity queries, and queries over XML.

Threshold and ranking queries were handled by a single server-side user defined function that evaluated each search term independently, calculated the total number of matches, and allowed results to be ordered by the number of matching terms. Queries on large text fields were handled in two steps. During pre-processing, the server scanned all text columns and created additional tables containing metadata to expedite these queries. During query execution, the baseline client translated queries over large text fields into new queries that searched these metadata tables.

5.5 Performance Monitor

A new feature built for Phase 2 was the measurement of system resources used by performer systems. The motivation for this work was to go beyond simply measuring query response time and throughput during the evaluation. In particular, system resource load data allowed us to inform eventual end-users on how different hardware might affect system performance (e.g., knowing that memory usage is a bottleneck allows a user to buy more RAM to compensate). As a side benefit, these monitoring tools enabled us to detect bugs in performer software during testing, like the memory leak shown in Figure 4.

To measure system resource load, we implemented two different automated monitoring systems. The first of these used the `collectd` daemon, which gathered statistics from system hosts and forwarded data to a central repository for collection and analysis. However, `collectd` could not accurately sample resource utilization at a rate greater than 1 Hz.

Knowing this limitation, we also architected a measurement system based on the `tcpdump` and `collectl` utilities. `tcpdump` captured packets traversing the network, and allowed us to measure bandwidth consumption. `collectl` measured CPU, RAM, and disk usage, and could do so at arbitrarily high frequencies (though when measured more frequently than 100 Hz, `collectl` itself consumes a non-negligible amount of resources and skews the metrics it is trying to capture). We built a Python program that consumed the raw output

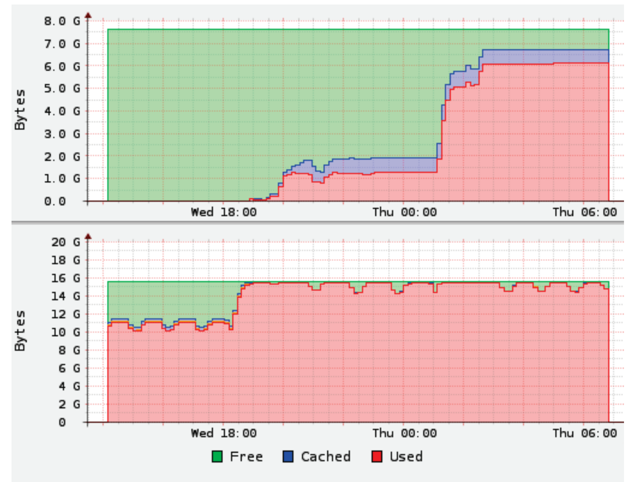

Figure 4: Capture of the swap (top) and memory (bottom) usage during a test. Spikes in these graphs correspond to the start of new tests. Machine crashes at about 7:00 am due to a memory leak.

from `collectl` and `tcpdump` and parsed relevant information into the SQLite integration database. As a result, queries into this database could overlay system resource usage with other test results and use `matplotlib` to produce visual performance trends.

5.6 Test Orchestrator

As discussed in Section 4.2, Phase 1 demonstrated the need for a more robust framework with which to orchestrate test execution. Test orchestration initially comprised various shell scripts with hardcoded test parameters; these were not robust to rapidly changing test requirements and required frequent manual modification during tests. Moreover, manually executing most of the process in Figure 1 was prone to operator error, which increased the risk of losing critical test data and made post-test analysis unnecessarily difficult.

The need for a flexible and structured test orchestrator was especially highlighted by the difficulty we had integrating with each performer's software prior to testing. While performers had access to our assessment framework in Phase 1, a deep understanding of each framework component was required to use it effectively. As a result, integration never occurred until the actual evaluation, which reduced our *test-time agility*. We needed a solution that would more effectively *automate the end-to-end test flow* and would simplify the interface between our framework and performer systems.

As illustrated in Figure 2, the test orchestrator allowed us to treat performer systems as *black boxes*. To this end, the orchestrator performed three high-level functions. First, test prerequisites were enforced on each remote host in the test, with any dependent files and packages automatically installed prior to executing a test case. Second, components were executed on the appropriate remote hosts via mechanisms that incurred *minimal system and network overhead*, required minimal software dependencies on remote hosts, and enabled *real-time monitoring* of each component's state. Third, all relevant test artifacts were *comprehensively cap-*

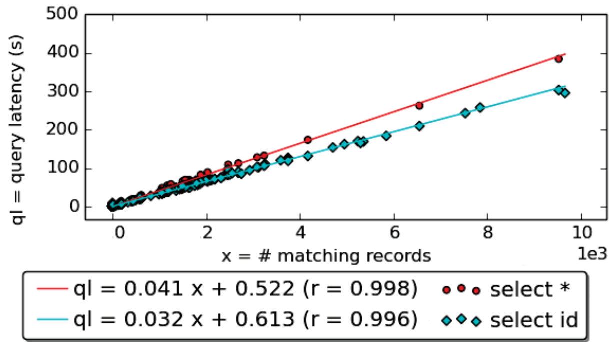

Figure 5: Automatically-generated graph displaying one performer's query latency as a function of the number of database records returned.

tured and archived, including the command-line output of all components and any generated logs.

We initially performed a survey of existing tools (Puppet[4], Chef[5], MCollective[6], and Ansible[7]) that could be the basis of a suitable test orchestrator. Unfortunately, these tools either required active daemons on remote hosts, were too young a technology to use at the time of our survey, or required us to build extensive wrappers around their platforms.

As no off-the-shelf product adequately met our needs, we built our own tool called `remote-runner`. `remote-runner` was a Python application based on `paramiko`'s SSHv2 implementation[8]. It only required two non-daemon packages to be installed on all remote hosts: OpenSSL[9] (to generate file hashes) and GNU Screen[10] (to create attachable sessions for each task). `remote-runner` provided a generalized interface with which users could define tasks using two custom Python classes: `Components` and `Muddlers`.

`Components` were individual tasks requiring remote execution. Performers would define `Components` for their client, server, and third-party software components, and we defined an analogous set for our baseline system. `Component` definition was meant to be very intuitive; each would minimally specify an executable, a list of arguments, a remote host, and a list of dependent files with their desired destinations. Additional attributes could be specified as the situation required, such as login credentials for remote hosts, the number of CPU cores to allocate for the executable, and more.

`Muddlers` were post-processors that received a list of `Components` and altered, composed, and sequenced them as needed. Performers did not have to compose `Muddlers`. However, we put considerable effort into developing `Muddlers` for test execution sequences. Our `Muddlers` simply accepted a set of performer `Components` and test parameters specified by

[4]puppetlabs.com/puppet/puppet-enterprise
[5]www.getchef.com
[6]puppetlabs.com/mcollective
[7]www.ansible.com
[8]github.com/paramiko/paramiko
[9]www.openssl.org
[10]www.gnu.org/software/screen

	b=1	b=2	b=3	b=4	b=5
a=0	0	13	74	87	90
a=1	86	89	96	96	97
a=2	86	90	96	99	100
a=3	88	90	100	100	100
a=4	89	94	100	100	100

Table 2: Using same performer data as Figure 5, but now compared to the baseline. Shows percentage of queries where performer $\leq a + b \cdot$ baseline, in seconds.

a test operator, then proceeded to execute entire test sequences. Our `Muddlers` internally defined special `Components` for our test harness, performance monitors, and anything else required to run a test, and would execute them on the appropriate hosts in the appropriate sequence to produce a set of parsable test artifacts. So long as a performer could properly define their `Components`, *automated end-to-end test execution* became trivial.

`remote-runner` met all the requirements we wanted to impose on a test orchestrator. Test execution was greatly streamlined, and it was considerably easier for both us and performers to integrate new systems into our test framework. Capturing all the data required to generate a report was greatly simplified, so much that a performer could theoretically generate reports independently.

5.7 Report Generation

With all of the test artifacts collected, the final step was to prepare a concise report that presented a system's correctness and performance characteristics in such a way that a human could quickly draw intelligent conclusions about each performer's prototype.

In Phase 1, all of our reports were composed by a mostly manual process aided by some analysis scripts that produced graphs and tables. The addition of any new data required a repetition of the entire process, and revisions to graphs and tables were invariably time consuming.

To simplify this task in Phase 2, we developed a tool that automatically and quickly (in less than 15 minutes) generated a detailed report describing the performance and correctness of each system from the SQLite integration database. The graphs and tables in the report were stored in easily-updated template files, and substituting in new data simply involved pointing the tool to an updated SQLite integration database. The SQLite database's contents allowed the report generator to easily graph and analyze the dependence of query latency on various factors; Figure 5 gives an example of such a graph. The SQLite database also facilitated comparisons to the baseline described in Section 5.4; Table 2 gives an example of such a comparison.

In addition to providing the basis for a final report after testing was completed, the report generator also granted us *test-time agility* by allowing us to quickly visualize and identify anomalies during testing without having to manually search through test artifacts. This allowed us to quickly diagnose bugs and configuration issues before repeating tests to capture better data.

6. CONCLUSION

In summary, we built a comprehensive framework for T&E on the SPAR program with a focus on upholding what we believe to be fundamental software T&E mantras. This framework not only effectively evaluated all nine performers' software under a variety of testing conditions, but was also designed and implemented with *black box* interfaces, *extensible* modules, and *reusable* components for future software assessment efforts.

Our assessment framework is open sourced at github.com/mitll-csa/sparta, along with detailed documentation describing the design and use of each component. We believe many readers can find reuse opportunities within our framework. With minimal reconfiguration, our entire framework could be used to benchmark a wide variety of database systems. Particular components, such as the data generator and knots library, could be used independently to address many synthetic data and high performance string processing needs.

We believe that a well designed assessment effort will, from the start, begin with consideration of the T&E mantras in Section 2. We hope that readers of this paper find our recommendations and exemplar assessment framework useful for their own efforts, and that these contributions can continue to progress the field of software and cyber system assessment.

7. ACKNOWLEDGEMENTS

The authors would like to acknowledge Oliver Dain for his design of the test harness architecture and knots data structure. Additionally, the authors thank Mike Depot, Tim Meunier, and John O'Connor for their valuable help.

8. REFERENCES

[1] H.-J. Boehm, R. R. Atkinson, and M. F. Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.

[2] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, volume 8042 of *LNCS*, pages 353–373. Springer, 2013.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[4] Department of Homeland Security. A roadmap for cybersecurity research, November 2009.

[5] A. Hamlin and J. Herzog. A test-suite generator for database systems. In *IEEE High-Performance Extreme Computing Conference*, 2014.

[6] IARPA. Broad agency announcement IARPA-BAA-11-01: Security and privacy assurance research (SPAR) program, February 2011.

[7] S. Peisert and M. Bishop. How to design computer security experiments. In *IFIP*, volume 237, pages 141–148. Springer, 2007.

[8] Project Gutenberg. `http://www.gutenberg.org`.

[9] M. Raykova, A. Cui, B. Vo, B. Liu, T. Malkin, S. M. Bellovin, and S. J. Stolfo. Usable, secure, private search. *IEEE Security & Privacy*, 10(5):53–60, 2012.

[10] US Census Bureau. Census 2000 5-percent public use microdata sample (PUMS) files. `http://www2.census.gov/census_2000/datasets/PUMS/FivePercent/`.

[11] US Census Bureau. Genealogy data: Frequently occurring surnames from census 2000. `http://www.census.gov/genealogy/www/data/2000surnames/`.

[12] C. V. Wright, C. Connelly, T. Braje, J. C. Rabek, L. M. Rossey, and R. K. Cunningham. Generating client workloads and high-fidelity network traffic for controllable, repeatable experiments in computer security. In *RAID*, volume 6307 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2010.

[13] Y. Yang. Evaluation of somewhat homomorphic encryption schemes. Master's thesis, Massachusetts Institute of Technology, 2013.