Quality Assurance of a Mobile Network Measurement Testbed Through Systematic Software Testing

Utkarsh Goel, James Espeland, Upulee Kanewala, and Mike P. Wittie
Department of Computer Science, Montana State University, Bozeman, MT USA 59717
{utkarsh.goel, james.espeland, upulee.kanewala, mwittie}@montana.edu

Abstract

The popularity of innovative mobile applications that offer services such as Web browsing, video streaming, online gaming, and collaborative communication puts utmost pressure on mobile application developers to ensure a high quality user experience. As such, the research and development communities have developed several networking testbeds that measure the performance of application traffic in production cellular networks. In this paper, we evaluate the quality of the source code of one of the mobile network measurement testbeds (MITATE) using multiple software testing techniques. Our extensive testing experience with MITATE's source code indicates that network measurement testbeds are complex in their functionality and require multiple software components to interact with each other for any given operation. We demonstrate that using multiple testing techniques results in different types of issues with the code under test. Finally, based on our results, we make changes to MITATE's source code and argue that MITATE, in production, offers high reliability and accuracy in executing network experiments.

Keywords: Black-box, mutation, testing, MITATE

1 Introduction

Software developers build interactive applications to attract and retain a large user base [25]. To ensure high engagement of users with their applications, developers employ sophisticated software development techniques that offer high quality user interfaces and easy navigation of the software [31]. As applications become popular, measurement of user-experience with the applications also becomes equally important. Specifically, applications that require network connections tend to work differently under different network environments [30]. Therefore, understanding the performance of applications under various network conditions (such as on wired networks, cellular networks, or Wi-Fi) enables developers to develop customized application logic based on how a given communication protocol performs in any one network condition. As shown in Table 1, applications such as online gaming, remote equipment operations, collaborative communication, and interactive artificial intelligence systems have different network requirements such as low end-to-end latency, low jitter, high throughput, and low packet loss. When these requirements are not met by the network, applications fail to offer a high quality experience to users. Therefore, developers should perform careful measurements of their applications and identify communication protocols that allow applications to perform reliably in networks with poor performance.

To perform large scale measurement of application performance, developers require tools that allow accurate analysis of the application performance under different network conditions. Although the networking research communities have developed several testbeds to rigorously measure application performance under different network scenarios [11, 12, 13, 28], developers often lack awareness as to which tool to use for testing the performance of their applications. Mobile Internet Testbed for Application Traffic Experimentation (MITATE) is the first testbed that allows developers to prototype application traffic, such as traffic from online games, social networks, and video streaming [30]. MITATE offers real-world performance testing of custom mobile applications, that network emulators and many other network measurement tools are not designed to offer [28]. In order to guarantee the appropriate functionality and operation of the MITATE software, in this paper we apply several systematic software testing techniques to the MITATE codebase. Specifically, we classify the four major contributions of this work as follows:

Novelty: In this paper we offer to the software testing community an understanding of how a mobile network measurement platform works, in terms of different software components that may be involved during execution of network experiments. We then provide an understanding of how various software testing techniques could be applied towards testing different components of a testbed written in several programming languages, as opposed to standalone applications.

Application Type	Network Latency	Jitter	Throughput	Packet Loss
Online Gaming	$100 - 500\,ms$ [27, 38]		$0.2 - 1.5 Mbps$ [1, 2, 5]	$< 1\%$ [33]
Interactive Communication	$100 - 300\,ms$ [4, 37, 24]	$< 20\,ms$ [37]	$0.1 - 8\,Mbps$ [3, 38]	$0.1\% - 2\%$ [37]
Remote Equipment Operations	$300 - 800\,ms$ [39]			$< 0.1\%$ [37]
Interactive AI Systems	$< 7 - 15\,ms$ [32, 21]			

Table 1: Application requirements for a high-quality user experience.

	Black-box Testing	Mutation Testing
Shell Script	No errors thrown for extra input.	Discovered a functionally equivalent mutant. *Mutation Score: 93.33%*
PHP Code	No human readable errors thrown for invalid input.	Identified misspelled variable names, several surviving mutants. *Mutation Score: 88.6%*

Table 2: Results from Black-box and Mutation testing.

Programs under test: To analyze the quality of MITATE software, we first identify code files that possess high risk in the operation of MITATE as a networking testbed. Specifically, we selected software components that allow developers to interact with the MITATE backend servers. The first selected component is a Shell script that presents MITATE's command line API and is the only interface through which developers can configure and upload network experiments for MITATE to execute on various mobile devices. The second component we select is a PHP code file that allows MITATE to rigorously validate the user experiment before it can be stored on MITATE servers and sent to mobile devices for execution.

In our testing of the above selected MITATE components, we perform black-box and mutation testing on both the Shell and PHP code. The Shell file has a McCabe Cyclomatic Complexity of 47 and consists of about 230 lines of code with 27 `if` conditions, two `for` loops, one `while` loop, and five function calls. The PHP file has a McCabe Cyclomatic Complexity of 82 and consists of about 250 lines of code with 39 `if` conditions, five `for` loops, five `while` loops, and three function calls [40]. Finally, by analyzing call and control flow graphs of the Shell and PHP code files, we also represent the complexity of a networking testbed in general.

Test Results: During our experience of employing black-box and mutation testing techniques to MITATE's source code, we make the following four observations:

- Our application of black-box testing on the Shell script discovered no faults in the code. However, we discovered that for invalid input arguments (more than three arguments) to the Shell command line API, the code does not generate any error message to the user which may help the user to correct the command.
- From our application of mutation testing on the Shell script using 15 mutants, we had only one surviving mutant that we identified as a functionally equivalent mutant due to how the logic of the Shell script is designed.
- We employed black-box testing on the PHP code and observed that for several invalid inputs, the PHP code did not throw a human-readable error. However, a system-library-generated error was thrown.
- Finally, our application of mutation testing on the PHP code using 44 mutants resulted in five surviving mutants of which two were due to misspelled variable names.

We summarize the results from our black-box and mutation testing in Table 2. Based on the observations about the code we made from our testing, we corrected MITATE's source code such that all of our test cases pass in secondary testing of the code using the same test suite. Moreover, our corrections to MITATE's source code left no errors unresolved that we identified during our primary testing.

Inferences Drawn: Based on our experience with different software testing techniques on MITATE's source code, our conclusions are twofold.

- We conclude that a single software testing technique may not be sufficient to detect and resolve different types of issues in the code. For example, although mutation testing is used for testing the completeness of the test suite, our mutation testing also lead us to detect bugs in the code.
- Our work in this direction allows the mobile application developer community to embrace more confidence in the reliability and quality of the MITATE testbed, which would result in development of mobile applications with high user satisfaction.

The rest of the paper is organized as follows. In Section 2, we provide an overview of MITATE as a network measurement testbed. In Section 3, we describe MITATE's source code. In Sections 4 and 5, we discuss our application of black-box and mutation testing on the Shell script and PHP code file respectively. In Section 6, we discuss some potential threats to validity of our work on testing a mobile network measurement testbed. Finally, we conclude in Section 7.

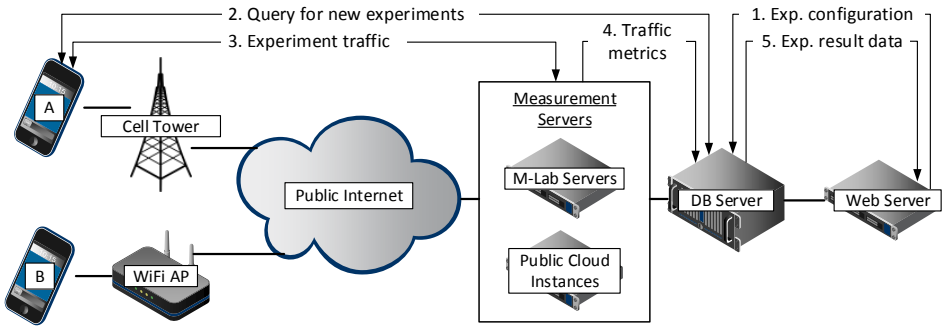

Figure 1: MITATE system architecture and steps to execute network experiments [30].

2 Overview of MITATE

Mobile Internet Testbed for Application Traffic Experimentation (MITATE) is the first-of-its-kind testbed that allows mobile application prototyping in production cellular networks [30]. MITATE is helpful to any developer interested in measuring the performance of their application on geographically distributed mobile devices in different cellular networks worldwide.[1]

We depict the architecture of the MITATE testbed in Figure 1. As shown in the figure, a mobile device connects to the public Internet via either the subscribed cellular network or a Wi-Fi network. The developer prepares network traffic definition in the form of a well-defined XML file. The traffic definition is comprised of the packet content that needs to be exchanged between the mobile device and the MITATE servers. The file also consists of criteria such as when and which mobile device should execute the experiment depending on device location, battery status, and network signal strength, among others. The developer then uploads the XML file to MITATE's database servers using a Web interface (Step 1). Upon successful upload, MITATE mobile clients poll for experiments for which they meet criteria defined in the XML file (Step 2). Next, (Step 3), MITATE mobile devices and servers exchange network packets. After the experiment finishes, MITATE servers record data representing network performance (in terms of latency, throughput, and jitter) in the database (Step 4). Finally, the measured network performance data is then available to the developer through a Web interface (Step 5).

In general, the MITATE testbed is comprised of four components: a mobile application, measurement servers, a command line API, and Web servers. An Android mobile client presents an interface to the users to install and run MITATE on their mobile devices as a background service. The measurement servers present interfaces for mobile clients to exchange network traffic, followed by recording of the measured data into the database. The command line API is used to access MITATE's Web interface through which experiments are uploaded and the measurement data is downloaded. The Web server allows mobile clients to probe for experiments for which the client meets the criteria.

The mobile application and measurement server code is written in the Java programming language, whereas the code for the Web server and command line API are written in PHP and Shell script respectively. For the purpose of this work, we perform testing on MITATE's most high-risk code, as opposed to testing MITATE's entirety of more than 9500 lines of code [10, 29]. Specifically, for the Web server, we tested the code that allows a user to evaluate the correctness of an XML file (representing the details of the experiment that needs to be executed between the mobile app and the measurement server) and then successfully upload the file to MITATE backend servers [8]. Finally, we tested a Shell script that allows developers to access MITATE's backend from a Linux terminal [7].

We argue that our decision to apply testing techniques to only the Shell script and PHP code is influenced by the fact that the Shell script and PHP code perform operations on input values directly defined by users. The other two modules, the Android mobile application and measurement server, do not interact with user-defined values and only execute operations once the data from the user has been correctly validated and stored into the database. Given that the majority of MITATE's operations depend on user-defined input, we therefore direct our software testing efforts to ensure that the code responsible for validating user input and correctly storing it in the database has no faults.

3 MITATE's Source Code

MITATE is a complex system, similar to other mobile network measurement testbeds, comprised of several software components and services that need to interact with each other in order to execute any operations.[2] Therefore, in this section, we delve into understanding

[1] As of March 2016, MITATE is being deployed on Google's Measurement Lab [6] and Android Play Store [15].

[2] For interested readers, we make the entire MITATE source code available to the public [29].

the complexity of different MITATE source code files using analysis of Control Flow Graphs (CFGs) for MITATE's Shell and PHP Code.[3]

3.1 Shell Script

MITATE's Shell script consists of approximately 230 lines of code, including 27 `if` statements, two `for` loops, one `while` loop, and five `function` calls [7]. The CFG for the Shell script has several basic blocks with varying number of outgoing paths to other blocks. Specifically, the Cyclomatic Complexity analysis of the Shell script indicates that the edge count is 119, the node count is 86, and the number of connected components is 7 [40]. Using the formula for McCabe's Cyclomatic Complexity, $M = E - N + 2P$, where E is the edge count, N is the node count, and P is the connected components count, we get a Cyclomatic Complexity score of 47 for the Shell script, indicating that the Shell script is complex and has high risks [26]. We argue that such a code file enables us to perform a well-thought systematic testing of the code and, in fact, evaluate the effectiveness of different testing techniques because of its complexity in how different code blocks interact with other blocks. We argue that given the number of predicates and loops in the code, MITATE's Shell script provides a unique opportunity to induce multiple mutations in the code when tested with the production MITATE backend code.

The Shell script is called with commands that consist of a single argument (`help`, `login`, `checkAvailableCredits`, and `logout`), commands that consist of two arguments (`init`, `validate`, `getExpCost`, `upload`, `makePublic`, `getExpStatus`, and `delete`), and a command that takes three arguments (`query`). The first argument specifies the operation that needs to be performed on the file or experiment ID specified by the second argument. For the two argument commands, the second argument is always a file name or an experiment ID. As shown in Figure 2, execution of the `help` command lists all commands that the Shell command supports along with appropriate arguments.

3.2 PHP Code

The PHP code file we select for testing allows the MITATE testbed to validate a user's uploaded XML file that describes the experiment to be executed [9]. If the XML file is validated successfully, the PHP code extracts the experiment data and uploads it into the database. The PHP file consists of approximately 250 lines of code, including 39 `if` statements, five `for` loops, five `while` loops, and three function calls [8]. Cyclomatic Complexity analysis of the PHP code indicates that the edge count is 189, the node count is 133, and the number of connected components is 13, resulting in a Cyclomatic Complexity score of 82 which indicates that it is a very high-risk code. Again, the number of connections between different code blocks indicates that the process of validating and uploading MITATE's XML input file is complex and thus motivates the need for using different software testing techniques.

[3]In a CFG, nodes represent blocks of continuous code and edges represent branches [22].

```
  $> ./mitate.sh help
mitate.sh <option>
- login
- init <output_filename>
- validate <XML_Filename>
- getExpCost <XML_Filename>
- checkAvailableCredits
- upload <XML_Filename>
- makePublic <experiment_ID>
- getExpStatus <experiment_ID>
- query <user_experiment_list.txt>
<output_filename>
- delete <experiment_ID>
- logout
```

Figure 2: List of Shell commands.

Given the complexity of selected components in MITATE's source code, our first goal is to identify whether or not the individual components generate correct results when given predetermined input values. Next, for any input value for which the code does not generate correct results, our goal is to identify specific lines of code that pertain to generation of incorrect results.

4 Black-box Testing

The first software testing technique we use to analyze the Shell script and the PHP code is black-box testing which creates test cases based only on the external specifications of the software [23]. The Shell script takes inputs in different forms, such as plain text and XML files. Therefore, our application of black-box testing is based on injecting different inputs to the Shell script and verifying the output using a predefined input and correct output values (which we refer to as the Oracle) [9]. For PHP code, we also inject varying XML files as input. These input XML files vary in terms of correctness of XML tags and definition of network experiments.

4.1 Shell Script

As stated earlier, the Shell script takes commands with one argument, two arguments, and three arguments. Therefore, we classify the domain of the input space for testing Shell script into 11 partitions, as shown in Table 3. To correctly partition the input space, we ensure that the union of all the partitions covers the entire domain of test inputs, as well as that the intersection of each pair of partitions results to null.

# Args.	Partition Number	Remark
0	P1	No arguments.
1	P2	Argument is valid.
	P3	Argument is invalid.
2	P4	Both arguments are valid.
	P5	First argument is invalid.
	P6	First argument is valid, the second is invalid.
	P7	First argument is valid, the second is empty.
3	P8	All arguments are valid.
	P9	First and second arguments are valid and the third argument is invalid.
	P10	First and third arguments are valid and the second argument is invalid.
>3	P11	First, second, and third arguments are valid.

Table 3: Input partitions for Black-box testing of Shell.

Using these partitions, we use a Combinatorial testing technique to generate different input argument commands and compare the output of these commands with the Oracle one-by-one [36]. Specifically, we generate test cases that contain all possible combinations of the input arguments to the Shell script. For example, the `login` command runs as a standalone operation. Therefore, for any other operation added to the `login` command, such as `upload` or `delete`, the Shell script must throw an error stating that the command was invalid. Using Combinatorial testing, we ensure that our test cases contain commands with arguments that are logically valid and invalid according to the Oracle.

Results from our application of black-box testing of the Shell script shows that for Partitions 1 through 11, we did not observe any difference in the output, when compared to the Oracle. However, for Partition 11, we found that for commands with number of arguments more than 3, no error is thrown. In general, no error is thrown for the inclusion of extraneous arguments.

For example,

```
$> ./mitate.sh login <random_data>
```

ignores the `<random_data>` and throws no error. We rewrote the logic of the command line API such that an appropriate error is now thrown when extraneous arguments are passed to the Shell script. The changes to the Shell script can be seen by visiting this reference [17]. After the changes to the code, we repeated the black-box testing with the same input partitions and observed that the output for all input partition values matched the values from the Oracle.

Partition Number	Remark
P1	Empty XML file.
P2	Valid XML file.
P3	XML file with invalid tags.
P4	XML file where integer values are replaced with character values.
P5	XML file where character values are replaced with integer values.
P6	XML file with extra tags.

Table 4: Input partitions for Black-box testing of PHP.

4.2 PHP Code

We now perform black-box testing of the PHP code by injecting errors into the input XML file. Based on the XML input specification [9], we identified six different input partitions for black-box testing of the PHP file, as shown in Table 4. We argue that since the input XML file could have any number of tags, it would be challenging to test all possible combinations of the XML tags, with combinations ranging from 2 tags to a significantly large number. Therefore, to generate a test-suite for testing the PHP code, we choose to manually perform Combinatorial testing where we create multiple combinations of XML tags that represent valid and invalid experiment configurations.

Our results from black-box testing of the PHP code indicates that when the input XML file is valid, the PHP code generates a valid output. However, when the input XML file is invalid, the PHP code does not validate the file but also does not output an error in a human-readable format. Based on these findings, we corrected the PHP code to output appropriate errors whenever the input XML file is invalid. We repeated the black-box testing with same input partitions on the new PHP code and observed that for all input partitions, the output of the PHP code matches with the Oracle.

5 Mutation Testing

The second software testing technique we apply to the Shell script and PHP code is mutation testing, where we inject errors (also called mutants) into the code and identify whether the output of the code with injected errors is different from the output of the original code [35]. If the output from a mutant is different for any test input, we say that the mutant is *killed*. Otherwise, it has *survived*. We use mutation testing to evaluate the effectiveness of the test-suite we created from the input partitions during black-box testing.

5.1 Shell Script

To perform mutation testing of the Shell script, we generated 15 mutants using a mutant clustering selection technique based on whether conditional

statements were un-nested, nested, or multi-nested inside other conditional statements [34]. Each of the mutants indicates an undesired change to one of the conditional statements in the code. Our decision to create mutants only in the conditional statements is due to the fact that the code performs all operations on user-defined input values and that there are no hard-coded values in the logic. Therefore, the conditional statements in the code are the only statements that could impact the overall output of the code. For interested readers, we host the various mutants at [14], where the file `mitate.sh` is the unaltered Shell script and files `m1.sh` through `m15.sh` are the 15 mutant files. In each mutant file, a change is made to only one of the conditional statements in the code as identified by a leading '#' sign with a comment that describes the change. Results from our application of the mutation testing on the Shell script show that all mutants, except `m4.sh`, are killed using the same test inputs we used during black-box testing. Since our test suite killed 14 out of 15 mutants, the mutation score is 93.33%.

The surviving mutant is a functionally equivalent mutant, because the Shell script authenticates the user before running any command. When the `login` command is run, the Shell script checks by default if the user is already logged in and a function that prints whether the user is valid or not is called with a parameter that *disallows* outputting the authentication message. When the `login` command is then executed, the function that prints whether or not the user is valid or not is called with a parameter that *allows* outputting the authentication message. So the function is called once with a parameter that disallows output and another time with a parameter that allows output. The surviving mutant is where a change was made to the code that determines whether to output the message or not. Since the function is called once with a parameter disallowing the output and another time with a parameter allowing the output, the same message is output once in both the original Shell script and the mutated Shell script. Therefore, it is not possible to tell which call of the function resulted in the output by just using mutation testing.

5.2 PHP Code

We choose to conduct mutation testing of the PHP code manually, as opposed to using an automated software tool such as Humbug [16]. Humbug automatically generates mutants from the input PHP source files. It then applies a given test-suite to the original source code and the mutated files to calculate how many mutants the test-suite kills. We argue that MITATE is a complex system that requires many different types of code files to be executed simultaneously. Therefore, we prefer to test MITATE's functionality in its real world deployment and thus we argue that simulating a virtual environment with Humbug may not reflect any code-related issues that may exist in the real world deployment of MITATE.

To perform mutation testing of the PHP code, we generated 44 mutants where each mutant represents an undesired change to one of each of the conditional statements. The conditional statements that were changed were either `if` statements or `while` statements. All of the `for` loops in the code were actually `foreach` loops where the code was looping through the contents of arrays. As such, the `foreach` loops did not contain conditional statements and were therefore not included in the scope of our mutation testing. Since we generated mutants for all of the conditional statements in the code, our technique to generate mutants implies that we also offer complete branch coverage with our test-suite. For interested readers, we host the different PHP mutant files at [18], where the `mitate_upload_experiment.php` file is the original file with no mutations and files `p1.php` through `p44.php` are the mutant files that we generated manually. For readability, we identify the mutant in each file by a leading '#' sign in the code. We also host our test input files comprised of the Shell scripts and XML input files for each of the 44 mutant files [19, 20].

To detect whether or not the various mutants were killed during our testing of the PHP code, we inspect the accuracy of the outputs from the MITATE Shell script and the database table corresponding to the operation. Our results show that our application of mutation testing killed all mutants, except for the mutants present in the files `p16.php, p30.php, p35.php, p43.php, and p44.php`. Specifically, we killed 39 out of 44 mutants during our first round of mutation testing, resulting in a mutation score of 88.6%.

Next, we inspected the code for the five surviving mutants. We identified that there were two different reasons as to why some mutants survived. Firstly, we identified that there were two minor bugs in the code due to spelling errors concerning variable names (as shown in `p30.php` and `p44.php` [18]). To resolve this first issue, we corrected the variable names. Secondly, we identified that some mutant code could not be reached due to how the logic was written in the PHP file. To resolve this issue, we refactored the code in the PHP files `p16.php` and `p43.php` so that the mutants become reachable while ensuring that the code functionality was not altered. We also added an `else` clause in `p35.php` to allow the corresponding mutant to be reached and ultimately killed. Finally, we repeated the mutation testing with the same input Shell scripts and XML files, during which we killed all the surviving mutants from the first round, resulting in a mutation score of 100%.

6 Threats to Validity

Internal validity: Given that some of the authors of this paper developed MITATE, we acknowledge that our test-suite selection may be biased towards our detailed understanding of the source code.

External validity: Although we highlight several features of a network measurement testbed in general, we do not claim that a similar infrastructure is used by other popular testbeds in the networking community. For example, while the MITATE mobile application runs as a background service on mobile devices, mobile applications of other testbeds run active experiments [28], and thus the source code would differ among these testbeds.

Conclusion validity: We argue that with our testing, we have increased confidence in the quality of MITATE as shown in Sections 4 and 5, however a threat to conclusion validity exists because we did not test the entirety of the MITATE source code.

Construct validity: Other software testing techniques may have uncovered issues in the tested code that we could not detect using black-box and mutation testing. However, we have no way of knowing this unless we apply other testing techniques. Finally, during our testing we assumed that the Internet connection was always available between different MITATE components. Therefore, another threat to construct validity exists because we did not analyze MITATE's quality when Internet connections were randomly disconnected.

7 Conclusions

Despite several years of research, mobile application performance remains a major concern for mobile developers. Therefore, developers are motivated to perform careful analysis of application performance in different network conditions. In this paper, we offer an understanding of how software testing techniques could be applied to test network measurement tools. Specifically, we apply sophisticated software testing techniques to a large mobile network measurement testbed (MITATE) to perform source code analysis and investigate its correctness. Results from our analysis illustrate that several software testing techniques should be used to uncover different types of issues with the code, such as misspelled variable names, equivalent mutants, and sub-optimal conditional statements. For example, in our testing, black-box testing alone did not uncover as many defects as we discovered using a combination of black-box and mutation testing. Finally, based on our testing results and modifications to the source code of MITATE, we argue that MITATE now offers even better reliability and accuracy in the execution of network experiments.

Acknowledgments

We thank Kanika Shah and the anonymous reviewers for providing us constructive feedback. We also thank National Science Foundation (NSF) for supporting this work through grants NSF CNS-1555591 and NSF CNS-1527097.

References

[1] Activision Bandwidth requirements for BO2? https://community.callofduty.com/thread/200620392.

[2] Call of Duty: Ghosts PC Minimum System Requirements. https://community.callofduty.com/community/call_of_duty/english/ghosts/blog/2013/10/23/call-of-duty-ghosts-pc-system-requirements, month = Oct, year = 2013.

[3] How much bandwidth does Skype need? https://support.skype.com/en/faq/FA1417/how-much-bandwidth-does-skype-need, month = Aug, year = 2014.

[4] Latency (audio). http://en.wikipedia.org/wiki/Latency_%28audio%29, month = Aug, year = 2014.

[5] Types of Broadband Connections. http://www.broadband.gov/broadband_types.html, month = Oct, year = 2013.

[6] Measurement Lab (MLab). http://www.measurementlab.net/, Sept. 2013.

[7] MITATE Shell Script - Command Line API. https://github.com/msu-netlab/MITATE/blob/master/WebServer/sample/mitate.sh, Mar. 2013.

[8] MITATE Validate Experiment. https://github.com/msu-netlab/MITATE/blob/master/WebServer/mitate_upload_experiment.php, Mar. 2013.

[9] Tutorial to MITATE: Mobile Internet Testbed for Application Traffic Experimentation. http://mitate.cs.montana.edu/sample/MITATE_User_Manual_v1.0.pdf, Mar. 2013.

[10] GitStats - MITATE.git. http://www.cs.montana.edu/~utkarsh.goel/MITATE-Stats/, Mar. 2014.

[11] Google Analytics. http://www.google.com/analytics/, Aug. 2015.

[12] Real User Monitoring. https://www.akamai.com/us/en/resources/real-user-monitoring.jsp, Aug. 2015.

[13] Real User Monitoring. http://www.keynote.com/solutions/monitoring/real-user-monitoring, Aug. 2015.

[14] Directory of Mutants of MITATE Shell Script. https://github.com/ugoel/CSCI591_SoftwareTesting/tree/master/shell_script_mutation, Mar. 2016.

[15] Google Play. https://play.google.com/store?hl=en, Apr. 2016.

[16] Humbug Mutation Testing Framework for PHP. https://github.com/padraic/humbug, Apr. 2016.

[17] Improved MITATE Shell Script - Command Line API. https://github.com/msu-netlab/MITATE/commit/490e7d9ad530544f636a0ba216ff4f34d864f7c6#diff-d4de63485f52c21e797c2e1260db5aa8, Mar. 2016.

[18] PHP Mutant Files. https://github.com/ugoel/CSCI591_SoftwareTesting/tree/master/php_code_mutation, Apr. 2016.

[19] PHP Mutant Files. http://mitate.cs.montana.edu/test/shell/, Apr. 2016.

[20] PHP Mutant Files. http://mitate.cs.montana.edu/test/xml/, Apr. 2016.

[21] M. Abrash. Latency – the sine qua non of AR and VR. http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/, 2012.

[22] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[23] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[24] G. Armitage. An experimental estimation of latency sensitivity in multiplayer Quake 3. In *International Conference on Networks*, 2003.

[25] Andreas Bernstom. How to attract users to your app and go platinum. http://venturebeat.com/2012/02/24/four-tips-on-how-to-attract-users-to-your-app/, Feb. 2012.

[26] Reg. Charney. Programming Tools: Code Complexity Metrics. http://www.linuxjournal.com/article/8035, Jan. 2005.

[27] Mark Claypool and Kajal Claypool. Latency and Player Actions in Online Games. *Commun. ACM*, 49(11), November 2006.

[28] U. Goel, M.P. Wittie, K.C. Claffy, and A. Le. Survey of End-to-End Mobile Network Measurement Testbeds, Tools, and Services. *Communications Surveys Tutorials, IEEE*, 18(1):105–123, Firstquarter 2016.

[29] Utkarsh Goel, Ajay Miyyapuram, and Mike P. Wittie. MITATE: Mobile Internet Testbed for Application Traffic Experimentation. https://github.com/msu-netlab/MITATE/, Mar. 2013.

[30] Utkarsh Goel, Ajay Miyyapuram, Mike P Wittie, and Qing Yang. MITATE: Mobile Internet Testbed for Application Traffic Experimentation. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 224–236. Springer, 2013.

[31] S. Halle, N. Bergeron, F. Guerin, and G. Le Breton. Testing Web Applications Through Layout Constraints. In *International Conference on Software Testing, Verification and Validation (ICST)*, April 2015.

[32] O. Hansen. The biggest problem in Augmented Reality: Latency. https://identifeye.wordpress.com/2013/01/03/the-biggest-problem-in-augmented-reality-latency/, 2013.

[33] E. Howard, C. Cooper, M. P. Wittie, S. Swinford, and Q. Yang. Cascading Impact of Lag on User Experience in Multiplayer Games. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[34] Shamaila Hussain. Mutation Clustering. http://www0.cs.ucl.ac.uk/staff/mharman/PastMScProjects2007/ShamailaHussain.pdf, 2008.

[35] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.*, 37(5), September 2011.

[36] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC, 1st edition, 2013.

[37] K. Lynch. Is your network ready to handle videoconferencing? http://www.techrepublic.com/blog/data-center/is-your-network-ready-to-handle-videoconferencing/, month = Jul, year = 2009.

[38] B. Mitchell. How Fast Does Your Network Need To Be? http://compnetworking.about.com/od/speedtests/tp/how-fast-does-your-network-need-to-be.htm.

[39] Reiza Rayman, Serguei Primak, Rajni Patel, Merhdad Moallem, Roya Morady, Mahdi Tavakoli, Vanja Subotic, Natalie Galbraith, Aimee van Wynsberghe, and Kris Croome. Effects of Latency on Telesurgery: An Experimental Study. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2005*, volume 3750 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005.

[40] Arthur H. Watson and Thomas J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric . http://www.linuxjournal.com/article/8035, Sept. 1996.