

MobiGUITAR

Automated Model-Based Testing of Mobile Apps

Domenico Amalfitano, Anna Rita Fasolino,
and Porfirio Tramontana, University of Naples Federico II

Bryan Dzung Ta and Atif M. Memon,
University of Maryland, College Park

// MobiGUITAR automates GUI-driven testing of Android apps, employing an abstraction of GUI widgets' run-time state. The abstraction is a scalable state machine model that, together with test coverage criteria, provides a way to automatically generate test cases. //



IT'S WIDELY believed that people adopt mobile platforms largely because of the apps they offer.^{1,2} So, app quality has become important.³ One of the most frequently used quality assurance techniques, even for mobile apps, is software testing. A relevant family of techniques and tools for automated testing of mobile apps focuses on app GUIs to find bugs. Such testing has been classified as random testing, model-based testing, and model-learning testing.⁴

Our previous research showed the feasibility of using model learning for testing Android mobile apps.⁵

Also, in previous research on conventional desktop applications, we developed *GUI ripping*, an automated model-based-testing technique.⁶ GUI ripping employs reverse engineering to generate test cases that execute directly on the software's GUI.

In addition, we developed test generation based on a reverse-engineered mobile-app model. Our original model—the event-flow graph (EFG)⁷—was stateless. This was a deliberate design decision to avoid state-space explosion. However, having a stateless model is no longer feasible because mobile apps are extremely

state sensitive. For example, consider the state-based life cycle of the Android Activity class, which forms the basis for Android app GUIs.

Moreover, we can no longer use the original test adequacy criteria based on EFGs. We need new, state-sensitive criteria. In addition, our test-case generators, also based on EFGs, are unusable; we need new ones that operate on state machines. Finally, when testing desktop applications, we never needed to handle security; we simply executed the test harness and application as the same user. However, most mobile platforms have enhanced security (for example, each Android app by default executes in its own sandbox). So, we need novel techniques for our reverse-engineering harness.

To overcome these challenges, we developed the MobiGUITAR (*Mobile GUI Testing Framework*) conceptual framework, which we implemented in a toolchain that executes on Android. MobiGUITAR models the state of the app's GUI, which helps us more accurately model mobile apps' state-sensitive behavior. This approach is consistent with ones proposed in the context of desktop Java apps⁸ and Web applications.^{9,10} Also, MobiGUITAR employs new test adequacy criteria based on state machines. A new test generation technique uses the models and criteria to generate test cases automatically. Finally, MobiGUITAR provides fully automatic testing that works with mobile platforms' security policies.

MobiGUITAR Overview

MobiGUITAR employs three primary steps: ripping, generation, and execution. In the following, we show how we used it to test Aard Dictionary (<https://github.com/aarddict>)

/android), a dictionary and offline Wikipedia reader.

Ripping

This step dynamically traverses an app's GUI and creates its state machine model. Because one of our goals is full automation, we create the model via GUI ripping. However,

generation, the ripper might view several GUI states as equivalent, using a given criterion, and merge them. In practice, this turns the tree into a directed graph because the ripper might encounter the same state in multiple ways. The ripper determines the equivalence between encountered GUI states on the basis of

Model-based testing
is a promising approach
for achieving better fault detection.

unlike our previous research, MobiGUITAR obtains a state-machine model of the GUI (not an EFG), and it uses algorithms better suited for mobile platforms.

Our ripper, an enhanced version of AndroidRipper,⁵ first launches the app in a given start state and obtains a list of events that can be performed on the GUI in this state. It adds this list, with each event as a separate task, to a task list, which it uses to fire events. The ripper removes an element in the task list and fires it. New states occur, and the GUI's focus changes as the events are fired. When the current state changes, the ripper obtains the list of new fireable events and appends it to the task list such that the path from the start state is prepended to each event. So, formally, a task is a sequence of events that always begins with an event that's fireable in the start state.

This process essentially realizes a breadth-first traversal of the app's GUI. During this process, a tree of GUI states can be maintained. In practice, the number of encountered states might be huge, making the tree and its traversal inefficient. For test

their constitutive objects' properties. Equivalent states comprise equivalent objects whose IDs and type properties have the same values.

Applying the ripper to Aard Dictionary yielded the state machine in Figure 1. For space considerations, we compacted the state and event IDs. The diagram demonstrates two main points. First, the user interaction space is quite flexible in that this machine has many paths and loops. For instance, the user can perform many event sequences switching between states a5, a45, a58, a46, and a61. The user can then go back to the start state; navigate to a3, a17, a53, and a2; and still go back to the start state, and so on. This flexibility creates a situation ripe for failures resulting from specific event sequences. Second, the diagram has a special shape—the exit. This isn't a GUI state; it shows that the app terminated via events e7, e8, e40, e68, or e88.

Generation

This step uses the model and test adequacy criteria to obtain test cases, each modeled as a sequence of GUI events. As you can imagine,

the number of all possible event sequences that can be executed on any nontrivial app's GUI is extremely large (in principle, infinite because of loops). The test generation strategy needs to sample from this space; our state machine allows just that.

We develop a pairwise edge coverage criterion. Conceptually, this means that all pairs of adjacent edges (events) must be exercised together. To this end, we create pairs of all edges in our state machine that are adjacent to a node. For each pair, we generate a test case that's a path in the state machine from the start state to the pair being covered. For example, a2 has four incoming edges (e1, e11, e12, and e13) and six outgoing edges (e7, e8, e9, e10, e11, and e12). This creates $4 \times 6 = 24$ pairs to cover. The test case (e1, e11, e8) covers two of these pairs: (e1, e11) and (e11, e8). For Aard Dictionary, we generated 678 test cases with 2,747 fireable events.

Execution

This step replays the tests. Our current implementation of the test generator outputs test cases in the JUnit format. Such test cases can detect crashes during the app's execution. A tester can enhance a test case by adding JUnit-like `assert` statements to check for functional errors.

For Aard Dictionary, 674 of the 678 test cases executed with no problems and covered 70 percent of the code. Of the 678 test cases, four detected a bug that led to an unhandled `IllegalArgumentException`.

A Demonstration

We selected four study subjects from Google Play (<https://play.google.com/store/apps>):

- Aard Dictionary 1.4.1;

FIGURE 1. The abstract state machine for Aard Dictionary, a dictionary and offline Wikipedia reader. The user interaction space is quite flexible in that this machine has many paths and loops. The exit isn't a GUI state; it shows that the app terminated via events e7, e8, e40, e68, or e88.

- Tomdroid 0.5.0 (<https://launchpad.net/tomdroid>), a note-taking app;
- Book Catalogue 3.8.1 (<https://github.com/eleybourn/Book-Catalogue/wiki>), a book-cataloging app; and
- WordPress Revision 394 of the alpha version for Android (<http://android.svn.wordpress.org>), an interactive client for creating, updating, and managing blogs saved on a WordPress server.

All the apps are open source projects developed and maintained by active communities of programmers.

To show our toolchain's bug detection capability, we automatically executed MobiGUITAR on the four apps; we generated and executed 7,711 test cases total. Several test cases revealed bugs; in all, we detected 10 bugs: the seven bugs that ripping had detected plus three new ones.

Table 1 provides details about these bugs. For the bug class, we use part of an Android bug classification¹¹ that distinguishes between concurrency bugs (interaction of multiple processes or threads), activity bugs (incorrect management of the activity life cycle), and other bugs (incorrect application logic implementation). The table shows the exception that was thrown and the ticket number we opened to report the bug.

Some bugs showed that Android applications might have incorrect

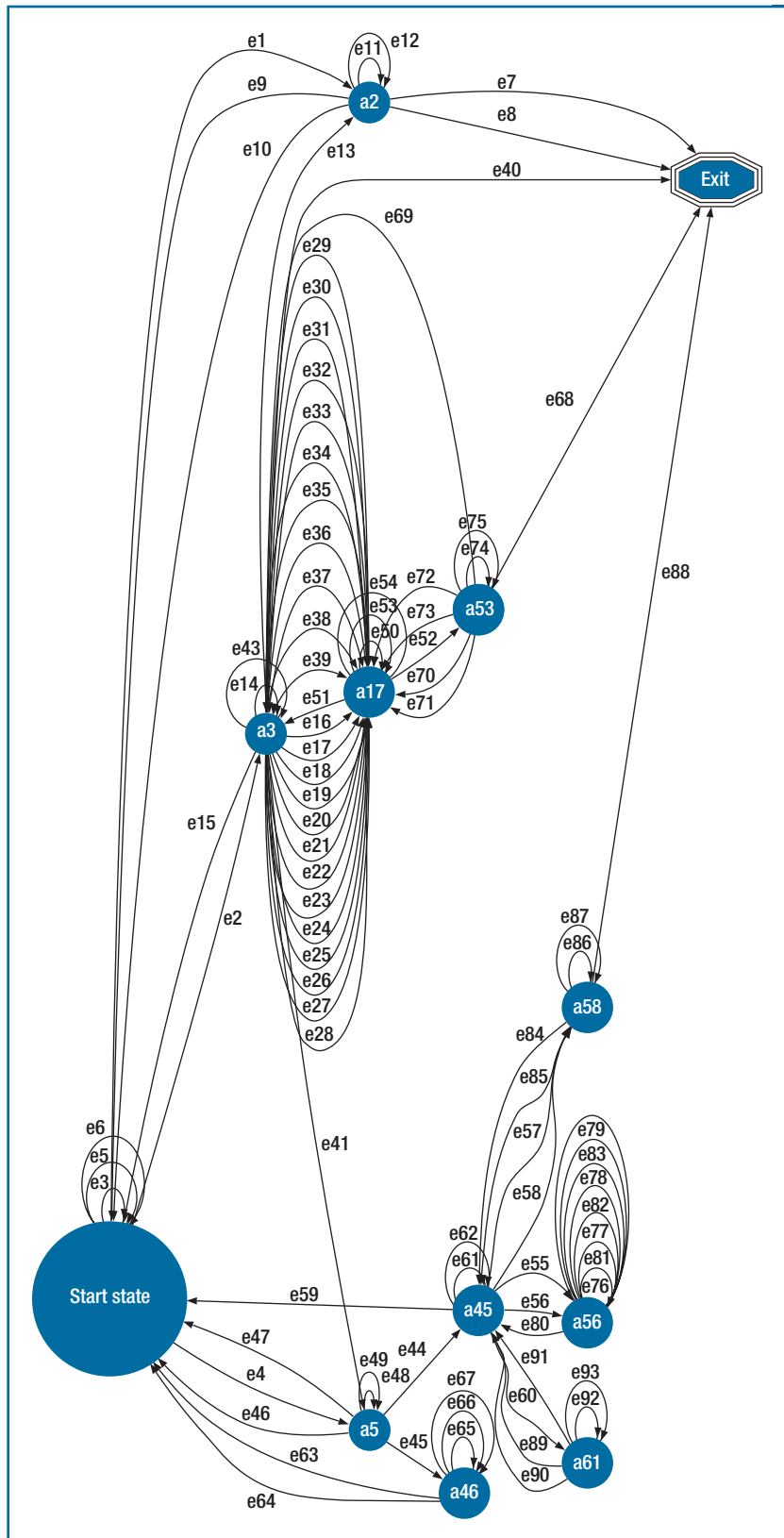


TABLE 1

Bugs detected by MobiGUITAR and the resulting failures.

| Bug ID | App | Bug class | Java exception | Ticket |
|--------|-----------------|-------------|--|---|
| 1 | Aard Dictionary | Activity | <code>IllegalArgumentException: View not attached to window manager</code> | https://github.com/aarddict/android/issues/44 |
| 2 | Tomdroid | Other | <code>IllegalArgumentException: Illegal character in schemeSpecificPart</code> | https://bugs.launchpad.net/tomdroid/+bug/902855 |
| 3 | Book Catalogue | Other | <code>CursorIndexOutOfBoundsException</code> | https://github.com/eleybourn/Book-Catalogue/issues/326 |
| 4 | Book Catalogue | Other | <code>NullPointerException</code> | https://github.com/eleybourn/Book-Catalogue/issues/305 |
| 5 | WordPress | Other | <code>StringIndexOutOfBoundsException</code> | https://android.trac.wordpress.org/ticket/206 |
| 6 | WordPress | Concurrency | <code>BadTokenException</code> | https://android.trac.wordpress.org/ticket/208 |
| 7 | WordPress | Concurrency | <code>NullPointerException</code> | https://android.trac.wordpress.org/ticket/212 |
| 8 | WordPress | Concurrency | <code>ActivityNotFoundException</code> | https://android.trac.wordpress.org/ticket/209 |
| 9 | WordPress | Other | <code>CursorIndexOutOfBoundsException</code> | https://android.trac.wordpress.org/ticket/207 |
| 10 | WordPress | Other | <code>NullPointerException</code> | https://android.trac.wordpress.org/ticket/218 |

behaviors due to incorrect management of their activities' life cycle. Activity components present a graphical user interface for each focused task the user can undertake. An activity instance passes through three main states: running, paused, and stopped. When it transits into and out of these states, it's notified through various callback methods (such as `onPause`, `onResume`, `onStop`, and `onDestroy`). These methods are hooks the programmer can override to do appropriate work when the activity's state changes. If the programmer fails to override or incorrectly overrides any of these methods, the app might show an incorrect behavior.

For example, Bug 1 resulted in the incorrect redefinition of `onPause()`.

It produced five crashes during the test cases' execution, owing to an unhandled `IllegalArgumentException`.

One sequence of events that caused the exception comprised

- e4 (open the Dictionary menu by pressing on the device's Menu button),
- e44 (click on a `MenuItem`),
- e57 (select a dictionary item from the list by a long click), and
- e88 (rotate the device).

After e4, e44, and e57, the handler `onItemLongClick` of e57 launched a thread to download a dictionary file from the device's SD (Secure Digital) card. It then instantiated a `ProgressDialog` onto the running activity to show the download progress. When e88

fired, the running activity was destroyed and then restarted with the updated configuration layout.

Unfortunately, the programmer didn't correctly override the dictionary activity's `onPause()` callback to save all the resources shown by the running activity, including the references to the working thread and its `ProgressDialog`. So, when the dictionary download ended and the `verified()` method tried to dismiss the `ProgressDialog`, an unhandled `IllegalArgumentException` crashed the app because the `ProgressDialog` was no longer attached to the window manager.

This bug showed us that testing the app through system events such as the orientation change can reveal this type of problem, which is frequent in Android applications.

TABLE 2

Tool feature comparison.

| Feature | Tool | | |
|--|---|------------------------|--|
| | MobiGUITAR | Monkey | Dynodroid |
| Types of fired events. | User events | User and system events | User and system events |
| Implemented testing technique. | Model learning and model based | Random | Model learning and random |
| Users can define input values. | Yes | No | No |
| Users can set a time delay between events. | Yes | Yes | No |
| Users can set preconditions. | Yes | Yes | No |
| Produced artifacts. | Crash reports Code coverage EMMA reports Executable JUnit test cases GUI sequences Finite-state-machine models Event sequences causing crashes | LogCat reports | LogCat reports Code coverage EMMA reports |

Bug 2 was due a sequence of events ending with the input of an incorrect URI into an `EditText` view. This event resulted in a `java.lang.IllegalArgumentException` that essentially was due to the lack of input validation in a method of the app. Currently, MobiGUITAR uses no automatic strategy to define the input values at run time. However, the tester can preliminarily configure the ripper to use a white list of specific input values. This bug showed us the necessity of investigating effective techniques for input value definition, such as the ones based on symbolic execution.¹²

Bugs 3 and 9 were due to incorrect code reuse. When programmers reuse code in different contexts, they might not test that code in the new scenarios, incorrectly assuming that the reused code will behave as well as in the original context. MobiGUITAR let us discover these bugs because it can test apps in different execution scenar-

ios, launching them from different preconditions.

Bugs 6, 7, and 8 were concurrency bugs that are typical in multithreaded software systems such as Android applications. We found them by configuring the tool to send events to the apps rapidly—that is, with a short delay between consecutive events.

Finally, Bug 8 depended on programming mechanisms that aren't traditional but are typical of Android apps. One example was the mechanism of the explicit intent to launch new activities at run time.

MobiGUITAR versus Other Tools

We compared MobiGUITAR to Monkey¹³ and Dynodroid,¹⁴ two tools for Android testing. We selected Monkey, a random-testing tool that comes with the Android Development Toolkit, because it's popular with Android developers. We chose Dynodroid because

it implements a variety of event-based-testing techniques. Like MobiGUITAR, both tools test apps by sending them sequences of events. We abstracted several tool features that are relevant in event-based testing, such as

- the types of fired events,
- the implemented testing technique,
- the ability to define input values,
- the ability to set a time delay between events,
- the ability to set preconditions, and
- the produced artifacts.

Table 2 reports the features and how each tool implements them. The tools' configurability features and produced artifacts differ considerably. MobiGUITAR produces several types of artifacts (such as crash reports, finite-state-machine models, GUI sequences, and executable JUnit test cases) that provide information

TABLE 3

Monkey's and Dynodroid's performance on unhandled exceptions.

| Bug ID | Found by | |
|--------|----------|-----------|
| | Monkey | Dynodroid |
| 1 | No | No |
| 2 | No | No |
| 3 | Yes | No |
| 4 | No | Yes |
| 5 | Yes | Yes |
| 6 | No | No |
| 7 | Yes | No |
| 8 | No | No |
| 9 | No | No |
| 10 | No | Yes |

useful for debugging. Monkey produces only an Android LogCat report; Dynodroid produces only an Android LogCat report and a code coverage EMMA (<http://emma.sourceforge.net>) report. So, their debugging support is limited.

Regarding tool configurability, only MobiGUITAR lets testers choose a white list of input values to be assigned with input widgets. Similarly, Monkey lets users choose event delays and put apps in some specific initial states. Dynodroid doesn't provide these features and always tests the app from the initial state of its having just been installed.


We configured Monkey and Dynodroid to test the four apps we tested earlier, with the same number of fired events. We analyzed the raised exceptions and the lines of code causing them. We matched them against the ones MobiGUITAR found that were associated with bugs. Unfortunately, we couldn't de-

bug all the other exceptions because of Monkey's and Dynodroid's poor debugging support.

Table 3 lists the evaluation results. Both tools found only three of the 10 exceptions; they didn't find any additional bugs. Monkey didn't detect the `IllegalArgument`Exception caused by Bug 1 owing to the randomness of its sent events. Dynodroid didn't detect this exception because it couldn't rotate the device. Neither Monkey nor Dynodroid found the crash related to Bug 2 because it's caused by specific values entered in a given `EditText` of the GUI.

Although Monkey lets users test apps by starting from a predefined initial state, it detected by chance just one of the unhandled exceptions concerning Bugs 3 and 9. Dynodroid didn't identify these exceptions because they're related to particular preconditions of the application under test. Dynodroid also didn't discover the crashes related to Bugs 6,

7, and 8 because they depend on a specific time delay between events. Although Monkey lets users configure such delays, it came across only one of the three exceptions. Monkey didn't detect the `NullPointerException` corresponding to Bugs 4 and 10.

Our results showed that MobiGUITAR generated test cases that were useful for detecting serious and relevant bugs in the apps. Moreover, this study showed that the combination of model-learning and model-based testing is a promising approach for achieving better fault detection in Android app testing. Because we're committed to the widest possible dissemination of our tool, we've made it available at <https://github.com/reverse-unina/AndroidRipper/wiki>. 

Acknowledgments

The US National Science Foundation partly supported this research under grant CNS-1205501.

References

1. I. Popa, "What Problems the Next BlackBerry Playbook Has to Solve in Order to Be Successful," 2012; <http://smartphone.straighttalk.com/what-problems-the-next-blackberry-playbook-has-to-solve-in-order-to-be-successful>.
2. M. Andrici, "Windows Phone: 70k Available Apps, but Are They Enough to Take on Android, iOS?," 2012; www.androidauthority.com/windows-phone-7-apps-67900.
3. M. Sama et al., "Context-Aware Adaptive Applications: Fault Patterns and Their Automated Identification," *IEEE Trans. Software Eng.*, vol. 36, no. 5, 2010, pp. 644–661.
4. W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," *Proc. 2013 ACM SIGPLAN Int'l Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA 13)*, 2013, pp. 623–640.
5. D. Amalfitano et al., "Using GUI Ripping for Automated Testing of Android Applica-



DOMENICO AMALFITANO is a postdoctoral researcher at the University of Naples Federico II. His research mainly concerns the reverse engineering, comprehension, migration, testing, and testing automation of event-driven software systems, mostly for Web applications, mobile applications, and GUIs. Amalfitano received a PhD in computer engineering and automation from the University of Naples Federico II. Contact him at domenico.amalfitano@unina.it.



ANNA RITA FASOLINO is an associate professor of computer science at the University of Naples Federico II. Her research interests are software engineering, software maintenance, reverse engineering, Web engineering, and software testing. Fasolino received a PhD in electronic and computer engineering from the University of Naples Federico II. Contact her at anna.fasolino@unina.it.



PORFIRIO TRAMONTANA is an assistant professor of computer science at the University of Naples Federico II. His research focuses on software engineering applied to mobile and Web applications. Tramontana received a PhD in computer engineering and automation from the University of Naples Federico II. Contact him at ptramont@unina.it.



BRYAN DZUNG TA is a fourth-year PhD student in the University of Maryland's Department of Computer Science. His research interests include empirical software engineering, software testing, program analysis, and security, focusing on GUI-based and mobile applications. Ta received his MS in computer science from the University of Maryland. Contact him at bryanta@cs.umd.edu.



ATIF M. MEMON is an associate professor in the University of Maryland's Department of Computer Science. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He has served on numerous US National Science Foundation panels and on program committees for events such as the International Conference on Software Engineering, International Symposium on the Foundations of Software Engineering, International Conference on Software Testing Verification and Validation, Web Engineering Track of the International World Wide Web Conference, Working Conference on Reverse Engineering, International Conference on Automated Software Engineering, and International Conference on Software Maintenance. Contact him at atif@cs.umd.edu.

tions," *Proc. 27th IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 12), 2012, pp. 258–261.

6. A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," *Proc. 10th Working Conf. Reverse Eng.* (WCRE 03), 2003, pp. 260–269.
7. A.M. Memon, M.L. Soffa, and M.E. Pollack, "Coverage Criteria for GUI Testing," *SIGSOFT Software Eng. Notes*, vol. 26, no. 5, 2001, pp. 256–267.
8. F. Gross, G. Fraser, and A. Zeller, "Search-Based System Testing: High Coverage, No False Alarms," *Proc. 2012 Int'l Symp. Software Testing and Analysis* (ISSTA 12), 2012, pp. 67–77.
9. A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Trans. Web*, vol. 6, no. 1, 2012, article 3.
10. D. Amalfitano, A. Fasolino, and P. Tramontana, "Reverse Engineering Finite State Machines from Rich Internet Applications," *Proc. 15th Working Conf. Reverse Eng.* (WCRE 08), 2008, pp. 69–73.
11. C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," *Proc. 6th Int'l Workshop Automation of Software Test* (AST 11), 2011, pp. 77–83.
12. S. Anand et al., "Automated Concolic Testing of Smartphone Apps," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, article 59.
13. "UI/Application Exerciser Monkey"; <http://developer.android.com/tools/help/monkey.html>.
14. A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," *Proc. 2013 9th Joint Meeting Foundations of Software Eng.* (ESEC/FSE 13), 2013, pp. 224–234.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.