metamorphic testing, cybersecurity, validation, software testing, cyberthreats, cyberattack, oracle problem, software security

Cover Feature

Metamorphic Testing for Cybersecurity

Tsong Yueh Chen and Fei-Ching Kuo, Swinburne University of Technology

Wenjuan Ma and Willy Susilo, University of Wollongong

Dave Towey, University of Nottingham Ningbo China

Jeffrey Voas, US National Institute of Standards and Technology

Zhi Quan Zhou, University of Wollongong

Metamorphic testing (MT) can enhance security testing by providing an alternative to using a testing oracle, which is often unavailable or impractical. The authors report how MT detected previously unknown bugs in real-world critical applications such as code obfuscators, giving evidence that software testing requires diverse perspectives to achieve greater cybersecurity.

Deploying inadequately tested software can have serious consequences for Internet and systems security, potentially resulting in "cyberspace catastrophes".[1] To avoid these effects, testers must adopt smarter testing techniques[1] for analyzing security-related functions. However, software testing is fundamentally challenged by the oracle problem,[2] when a test oracle—the mechanism that testers use to determine the correctness of test-case execution results—is not available or cannot be practically applied. Most software-testing techniques assume that an oracle is available, but that assumption is not always valid when testing complex applications, such as compilers, search engines, and software with diverse cryptographic algorithms. In these cases, an oracle is often unavailable or is theoretically available but too expensive to be practical.

The oracle problem also exacerbates negative testing—testing a program's behavior with invalid inputs—because testing with this type of input often means that the outputs are unpredictable or expensive to verify. Worse still, resource constraints might mean that testers skip negative testing, potentially allowing security holes to persist into the released software.[1],[4]

Metamorphic testing (MT) addresses this problem by providing a new testing perspective. Instead of focusing on each output's correctness for a program under test (PUT), MT looks at metamorphic relations (MRs)—how the inputs and outputs of multiple PUT executions relate. MRs include but are not limited to identity relationships.[9] Even if a test case does not reveal a failure, testers can use it to generate follow-up test cases by referring to selected MRs, and further test the PUT automatically. Because MT looks at MRs instead of individual outputs, software testing and analysis does not require an oracle, hence alleviating the oracle problem.[5]–[10]

Although MT's fault-detection performance might be slightly lower than an oracle's (which is why we say "alleviates" the problem), MT has been applied to test various applications, from numerical programs (such as those that perform scientific computation) to nonnumerical programs (such as search engines),[10] and has proved highly effective in detecting failures.[8],[11],[12]

To evaluate the effectiveness of MT in detecting bugs in real-world critical applications with no oracle, we used MT to test code obfuscators—software that transforms a program's original code into an

Rather than focusing on each obfuscated program's correctness, we inspected the MRs among multiple obfuscated programs.

We also evaluated MT's effectiveness in detecting Web failures by using it to test the login page of the National Australia Bank (NAB) for compatibility problems between the website and the client side. In this evaluation, we used a simple MR and still found problems.

In both the code obfuscator and Web tests, MT detected previously unknown faults. To the best of our knowledge, this article is the first report on the testing of code obfuscators' functional correctness and the detection of actual bugs in them.

Effects of the Oracle Problem

The testing of certificate-validation logic in SSL/TLS implementations illustrates the dilemma caused by the oracle problem.[3] If the PUT accepts a nontrivial test certificate, how can testers be sure that it is indeed valid? If the PUT rejects the certificate, how can they know whether or not the reason given for rejection is actually correct? As some researchers point out, determining test-certificate validity manually is not practical in large-scale testing, and automating the procedure "essentially requires reimplementing certificate validation, which is impractical and has high potential for bugs of its own."[3]

In their work on testing certificate-validation logic,[3] the researchers obtained several independently implemented programs for X.509 certificate validation, and could thus compare the programs' outputs for the same input certificates and note any discrepancies. However, even if discrepancies were detected, it might not be easy to know which program is correct—actually, all programs could be incorrect. In some situations, multiple implementations of the same specification cannot be obtained, and the oracle problem becomes more serious. In code obfuscator testing, for example, the tester must determine if the input (original) code and the output (obfuscated) code are equivalent, which can be extremely difficult.

The oracle problem also discourages testers from attempting fuzz testing, or *fuzzing*—an important negative-testing technique. Fuzzing subjects the PUT to invalid, random, or semirandom inputs. Because fuzzing is conceptually simple and easy to implement (through fuzzers), and can cause the PUT to crash in unexpected ways, fuzzing has a potentially high benefit-to-cost ratio and is thus recognized as an efficient automatic testing technique for detecting software and network vulnerabilities.[1],[4].

The oracle problem is a major challenge for fuzzing because verifying the output for large amounts of random or semirandom input data is extremely difficult, if not impossible. Rather than attempt this verification, fuzzing looks only for crashes or some other undesirable PUT behavior, and millions of test cases might be executed before a crash.[4] Moreover, many bugs such as logic errors[5] do not crash the PUT, but instead produce incorrect output—a failure type that is much more difficult to detect. The notorious Heartbleed bug,[1] for example, does not cause a crash and is therefore undetectable with simple fuzzing.[4]

How Metamorphic Testing (MT) Works

To illustrate how MT works, consider a PUT that implements the sine function. An MR for that PUT might be $\sin(x) = \sin(180 - x)$ and a test case might be $t = 32.875$. An output of, say, 0.543 might be hard to verify without an oracle. Regardless of whether or not an oracle is available, MT suggests a possible follow-up test case, $t' = 180 - 32.875$. After taking rounding errors into consideration, if the two outputs are not equal, then MT has revealed a failure.

The detection of compiler bugs is an example of MT application. Compiler correctness is extremely important because some of the programs being compiled might perform critical functions. Researchers at UC Davis won a distinguished paper award[13] for their compiler testing study, which was "based on a particularly clever application of metamorphic testing."[14] Their MR is a special instance of the following:

arbitrary input *I*, record code-coverage information with respect to *P*, and create *P*′ by randomly pruning some unexecuted statements from *P*. A compiler bug is reported if the output of *O*′ on *I* has changed. Researchers documented 147 confirmed, unique bug reports for GCC and LLVM alone.[13]

Detecting Obfuscator Bugs

Obfuscators are important in protecting confidential software elements, but bugs can be difficult to find, even with advanced compiler-testing techniques.[15] Our case study aimed to test well-known, real-world obfuscators using MT with diverse MRs and a small set of test cases.

Obfuscators tested

We tested four obfuscators:

- Cobfusc (http://manpages.ubuntu.com/manpages/hardy/man1/cobfusc.1.html) is an open source Linux utility that makes a C source file unreadable, but compilable.
- Stunnix CXX-Obfus is a commercially available C/C++ obfuscator, which was previously tested by other researchers with no detected failures.[15] CXX-Obfus users include the US Army and Fortune 500 companies.
- Tigress (http://tigress.cs.arizona.edu) is a freely available (but not open source) C obfuscator developed at the University of Arizona that supports novel defenses against both static and dynamic reverse engineering.
- Obfuscator-LLVM (https://github.com/obfuscator-llvm/obfuscator/wiki) is an open source tool in the LLVM compilation suite. Given a C source program, Obfuscator-LLVM outputs obfuscated and compiled binary code. Obfuscator-LLVM users include Adobe Systems, Apple, Intel, and Sony.

Metamorphic relations (MRs)

MT tests programs by referring to predefined MRs. Because programmers can make a variety of mistakes, we believe that a collection of diverse MRs will detect more faults than a single MR. For our case study, we defined four MRs (counting two versions of the first MR).

MR_1. The first MR states that, if two different source programs (P_1 and P_2) are functionally equivalent, their obfuscated versions ($O(P_1)$ and $O(P_2)$) will also be functionally equivalent and, therefore, the compiled obfuscated executable programs, $C(O(P_1))$ and $C(O(P_2))$, should have equivalent behavior—the same outputs for the same inputs.

Testing based on this MR required generating equivalent source programs P_1 and P_2 either by using a separate tool, such as a script written by the tester, or by using the obfuscator itself. We denoted the first strategy by $MR_{1.1}$ and the second by $MR_{1.2}$. For one $MR_{1.1}$, we defined P_1 as `If (condition) {do A} else {do B}` and P_2 as `If (not(condition)) {do B} else {do A}`. For $MR_{1.2}$ we used the obfuscator on P_1 to generate the (supposedly) equivalent program $O(P_1)$. We ran this obfuscation twice to obtain $P_2 = O(O(P_1))$.

MR_2. This MR states that an obfuscator should generate behaviorally equivalent programs for the same input program, regardless of the obfuscator's execution environment. In this study, we defined that environment as time: for the same input program, the obfuscator should generate behaviorally equivalent programs regardless of when the obfuscator is run.

MR_3. The final MR differs from the others in that it looks at obfuscated source code without compiling it. In contrast, $MR_{1.1}$, $MR_{1.2}$, and MR_2 focus on the compiled obfuscated programs' behavioral equivalence when run on the same inputs. MR_3 checks whether obfuscation rules have been applied consistently each time

check if outputs are consistent. For example, if a variable name in program *P* was obfuscated when the obfuscator ran yesterday, then the same variable name should still become obfuscated when the obfuscator is run today.

Sample failures and other detected issues

We tested the obfuscators using 500 randomly generated source test cases (C programs), finding bugs or other issues in every obfuscator under test. Except for Obfuscator-LLVM, we tested all four of the MRs we defined on all four obfuscators. (Obfuscator-LLVM generates only obfuscated binary code without showing the obfuscated source code, so $MR_{1.2}$ and MR_3 are not applicable to its testing.) We also found that different MRs detected different kinds of issues; for brevity, we describe only one issue for each MR.

$MR_{1.1}$. Figure 1 shows excerpts of input files that revealed a failure in Tigress (version: Linux x86_64-unstable revision 1676) when tested against this MR. The test case, P_1, has two integer variables *i* and *j*, each of which is assigned an initial value. If *i* is greater than *j* then *i* is set to *i* − 10, otherwise *i* is set to *i* + 10. Finally, the value of *i* is printed. The upper left box of Figure 1 shows the essential part of the P_1 code. The corresponding code of an equivalent program P_2 (the follow-up test case) is shown in the lower left box. $O(P_1)$ and $O(P_2)$ are the obfuscated codes of P_1 and P_2, the essential parts of which appear in the upper and lower right boxes of Figure 1. In a metamorphic test, $O(P_1)$ and $O(P_2)$ were compiled into executable programs $C(O(P_1))$ and $C(O(P_2))$, which were then run on the same input, and their outputs compared. MT detected that the outputs of $C(O(P_1))$ and $C(O(P_2))$ were different, thereby detecting a bug in Tigress.

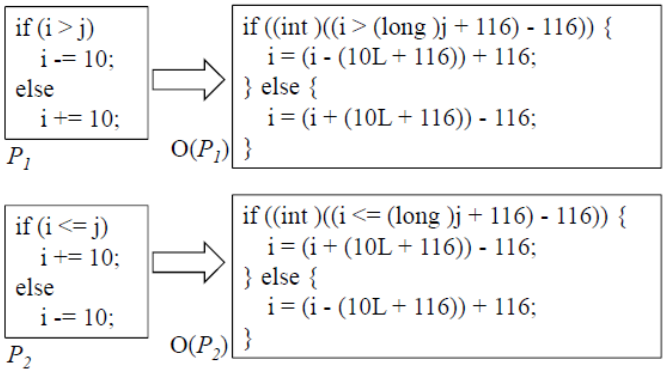

Figure 1. Tigress failure detected against metamorphic relation (MR) $MR_{1.1}$. P_1 and P_2 were the metamorphic testing (MT) test cases, which were obfuscated into $O(P_1)$ and $O(P_2)$. After compiling $O(P_1)$ and $O(P_2)$, and running the executable programs on the same input, MT detected differences in the outputs, which signaled a bug.

As Figure 1 shows, Tigress incorrectly obfuscated the P_1 statement `if (i > j)` into the $O(P_1)$ statement `if ((int )((i > (long )j + 116) – 116))`. In C, the expression `(i > (long )j + 116)` is evaluated to either true (1) or false (0), so the expression `((i > (long )j + 116) – 116)` is evaluated to either −115 or −116, both of which are nonzero. In C, any nonzero value means true. Consequently, the `if` statement of $O(P_1)$ will always take the true branch, and the false branch will be unreachable, which means that $O(P_1)$ is not equivalent to P_1. Likewise, the false branch of $O(P_2)$ will also be unreachable. When testing against $MR_{1.1}$, we ran $C(O(P_1))$ and $C(O(P_2))$ on the same input—*i* = *j* =1,000. After the `if` statement, $C(O(P_1))$ set *i* to *i* – 10, (that is, 990), but $C(O(P_2))$ set it to *i* + 10 (that is, 1,010), thus generating different outputs. As a result, testing against $MR_{1.1}$ revealed a bug in Tigress.

Arguably, we could have detected the bug without MT, by compiling P_1 into $C(P_1)$, running $C(P_1)$ and $C(O(P_1))$ on the same input, and comparing their outputs. However, this conventional testing method can

less than or equal to *j*'s initial value. In contrast, MT guarantees bug detection because the outputs of $C(O(P_1))$ and $C(O(P_2))$ will always be different regardless of these initial values. In this example, therefore, MT appears superior to conventional testing methods, emphasizing the need to test from diverse perspectives.

$MR_{1.2}$. To test Cobfusc (package cutils version 1.6), we ran source test case P_1, which included the statement

```
int k = 20; //Rz5Wq3OCvuqsA30uaEY0Evc95AIn
```

We then recursively called Cobfusc to construct P_2 as $O(O(P_1))$. We expected $O(P_1)$ and $O(P_2)$ to be equivalent, but surprisingly, $O(P_2)$ could not pass through the compiler because the obfuscator had incorrectly moved the comment Rz5Wq3OCvuqsA30uaEY0Evc95AIn from its original line into a separate new line without the // :

```
int k = ((5*(1*1+0)+2)*((2*(1*1+0)+0)*(1*(1*1+0)+0)+0)+(3*(2*1+0)+0)); //
Rz5Wq3OCvuqsA30uaEY0Evc95AIn
```

Thus, $MR_{1.2}$ had detected a bug in Cobfusc.

MR_2. MR_2 states that, when an obfuscator runs at different times for the same program, the output programs should be equivalent. Given a C source program, Obfuscator-LLVM can be enabled by running Clang, LLVM's front end compiler, to obtain obfuscated and compiled binary code. In Figure 2a, line 1 shows a source program PBP.c compiled by Clang with command line parameters enabling obfuscation. The compiled obfuscated executable program (a.out) was run in line 2 with an input of 10000022, producing the output 10000022 in line 3. Figure 2a shows a behavioral inconsistency in the compiled obfuscated executable programs, which produced an output of 10000022 in lines 3 and 9, and 14195494 in line 6, with the same procedure of obfuscation, compilation, and execution. Figure 2b shows that when Clang was run without enabling obfuscation, the compiled executable programs produced identical outputs (lines 3, 6, and 9).

```
$ clang -mllvm -bcf -mllvm -boguscf-loop=3 PBP.c
$ a.out 10000022
10000022
$ clang -mllvm -bcf -mllvm -boguscf-loop=3 PBP.c
$ a.out 10000022
14195494
$ clang -mllvm -bcf -mllvm -boguscf-loop=3 PBP.c
$ a.out 10000022
10000022
```

(a)

```
$ clang PBP.c
$ a.out 10000022
14195494
$ clang PBP.c
$ a.out 10000022
14195494
$ clang PBP.c
$ a.out 10000022
14195494
```

(b)

Figure 2. Results of Clang compilation of the PBP.c program to generate binary code a.out when obfuscation was (a) enabled and (b) disabled. In (a), Clang did not generate behaviorally equivalent binary code. Clang ran the compiled obfuscated executable program (a.out) in line 2 with an input of 10000022, producing the output in line 3 of 10000022. After the same procedure of obfuscation, compilation, and execution (lines 4 and 5), the output was 14195494 in line 6. In (b), Clang consistently produced an output of 14195494. (Results are from Obfuscator LLVM version 3.4)

The issue was caused by five PBP.c statements: int i; int j; i=atoi(argv[1]); i=i+j; printf("%d\n",i); *i* is initialized with the input value (10000022 in Figure 2) and then updated by the statement i=i+j;—*j* is used without initialization and, therefore, the value of *i* after this statement cannot be predicted. This value of *i* is printed by the printf statement. Figure 2a does not indicate whether the output 10000022 or 14195494 is wrong, but instead shows that the executable programs generated for the same input program are not behaviorally

We further investigated this issue by again using Clang to compile PBP.c but without enabling the obfuscation function. Figure 2b shows the result. The compiled executable programs (a.out) consistently produced 14195494, regardless of the number of times we ran the compiler.

MR_3. MR_3 involves checking that obfuscated source files based on the same input source file are consistent. We ran CXX-OBfus twice on the same source code to generate two obfuscated output files. Figure 3a shows an excerpt of the source code before obfuscation; Figures 3b and 3c show an inconsistency in the obfuscated code after one and two runs. Although the obfuscation rules of Stunnix are unknown, any inconsistent behavior is undesirable.

```
1 #include <stdio.h>
2 int j = 1908;
3 int k = 1662;
4 int m = 1734;
5 int n = 468;
6 int p = 1046;
7 int q = 613;
```

(a)

```
1 #include <stdio.h>
2 int j = (0x1cc8+2138-0x1dae);
3 int k = (0x734+890-0x430);
4 int m = (0x9e7+132-0x3a5);
5 int n = (0x11e1+3746-0x1eaf);
6 int p = 1046;
7 int q = (0x30b+5768-0x172e);
```

(b)

```
1 #include <stdio.h>
2 int j = (0x1e12+3135-0x22dd);
3 int k = (0xcb8+260-0x73e);
4 int m = (0x1da8+1116-0x1b3e);
5 int n = (0x239b+1251-0x26aa);
6 int p = (0x90f+4654-0x1727);
7 int q = (0x6d1+6323-0x1d1f);
```

(c)

Figure 3. Detecting an inconsistency in the Stunnix CXX-Obfus obfuscator (version 4.2). If the (a) original source code is obfuscated in one run, it should be obfuscated in other runs so that any confidential information is always protected. In (b), the results of the first run, line 6 is the same as line 6 in the original code, but in (c), the results of the second run, line 6 is obfuscated.

Detecting Web Failures

In our evaluation of MT for website validation, we identified a simple MR from the perspective of website users rather than developers: When different users log onto a (local) computer using different usernames, they should always be able to follow the same steps to navigate to the website and click on the Internet banking login button—changing usernames on a local computer should not affect access, as long as all the users' account settings are standard. During testing, we automatically captured screenshots of different user sessions and compared them to identify potential issues in the website GUI. The use of screenshot comparisons in Web testing has been documented as a viable method.[16]

Figure 4 shows an Internet banking login failure that our test driver automatically detected. Figure 4a is the result of a tester logging onto the local computer, an Acer Chromebook with standard settings, using the default guest account. The user successfully opened the NAB website using the Chrome browser, and successfully clicked on the red Login button. However, when the tester logged onto the same Acer Chromebook with a different username and then used the same browser to open the same website (the environment settings were all standard), an Internet banking login failure occurred.

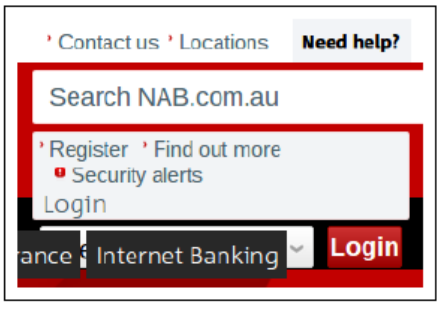

Figure 4. Detection of a login failure in the National Australia Bank (NAB) Internet banking website. (a) Normal Internet banking login with local "guest" username and (b) failure with a local non-guest username. When the tester logged onto the same Acer Chromebook with a nonguest username and employed the same browser, the website GUI did not display correctly, and the tester could not click on either the Login button or the grey Login link.

Although developers might argue that this is not a verification bug—because the NAB website was not designed to support this platform and configuration—it is obviously a validation problem from the user's perspective.

MT and Negative Testing

MT has considerable potential to guide negative testing to detect security vulnerabilities. To explore this idea, we used MT to detect the infamous Heartbleed bug.

The Heartbleed bug

The Heartbleed bug is probably the most widely known cybersecurity breach in recent years,[1] appearing in the OpenSSL implementation of the Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension specified in RFC 6520. As shown in Figure 5, a Heartbeat protocol message consists of type, payload, padding, and payload_length, with the statement `opaque payload[HeartbeatMessage.payload_length];` meaning that the actual payload length must equal `payload_length`. When implementing this protocol, the programmer assumed that the relationship between `payload` and `payload_length` would always hold true and therefore did not include any bounds-checking code. The core OpenSSL developer who reviewed the implementation had the same wrong assumption and also did not find the bug. Both made the same common mistake: they overlooked the possibility that some parameters can take a value outside the expected range.

4. Heartbeat Request and Response Messages

The Heartbeat protocol messages consist of their type and an arbitrary payload and padding.

```
struct {
   HeartbeatMessageType type;
   uint16 payload_length;
   opaque payload[HeartbeatMessage.payload_length];
   opaque padding[padding_length];
} HeartbeatMessage;
```

Figure 5. Excerpt from Section 4 of RFC 6520 (http://tools.ietf.org/html/rfc6520). MT holds great potential for guiding negative testing to verify implementations of security-related protocols like the Heartbeat protocol.

Detecting the Heartbleed bug using MT

To identify MRs to test implementations of the Heartbeat protocol messages (Figure 5), the tester will typically ask, "What if I change some of the parameter values?" Suppose that the source test case is t = ($type_1$, $length_1$, $payload_1$, $padding_1$), where $type_1$, $length_1$, $payload_1$, and $padding_1$ represent concrete parameter values. To identify an MR, the tester will ask:

- What if I change $type_1$ to a different value?
- What if I change length to a different value?

- What if I change $padding_1$ to a different value?
- What if I change two or more parameters?

Asking these questions will lead the tester to think beyond the normal range of parameter values or value combinations, leading to negative testing. For example, question 2 should stimulate thoughts that payload_length might not necessarily equal payload, and prompt the construction of a follow-up test case *t*′ that increases the value of payload_length while keeping the other parameters unchanged. The MR will require different behavior for *t* and *t*′ (per RFC 6520): the PUT should return a normal message for *t*, but discard *t*′. When the Heartbleed bug is tested against this MR, it will have the same behavior for both *t* and *t*′ by always returning a message (buffer), which will violate the MR, hence revealing a failure.

The Heartbleed bug could also be detected by using a fuzzer in conjunction with some dynamic analysis tools performing run-time monitoring. However, these tools are restricted to predefined memory error types,[1] giving MT a distinct advantage in negative testing because MT can detect system crashes (in which MRs will always be violated) and other error types, such as incorrect or inconsistent behavior caused by logic errors.

Previous work has shown the feasibility of combining MT and fuzzing.[7] When testing Microsoft Live Search, a random string GLIF was issued, for which the search engine returned 11,783 results. Owing to the sheer volume of data on the Internet, it was difficult to assess the correctness of the results. Nevertheless, by referring to an MR, a follow-up test case GLIF OR 5Y4W was generated (where OR is the Boolean operator, not a search term), and the search engine returned zero results. This was obviously a failure, as the number of Web pages containing either the string GLIF or the string 5Y4W should be no less than 11,783. When generating this incorrect result, the search engine did not crash. The failure, therefore, could not be detected by fuzzing or dynamic analysis (or a combination of the two). Nevertheless, MT detected the failure by comparing the outputs of multiple executions.

We have shown that MT can help alleviate the oracle problem, detect Web failures, and enable negative testing of security-related functionality and behavior. In our evaluations, MT successfully revealed real-life bugs that other testing methods failed to detect, not only because MT is possible without an oracle, but also because MT is based on a perspective that conventional testing techniques do not use. MT is not necessarily always better than other testing methods, but it does show the effectiveness of conducting testing from diverse perspectives—thus mirroring the diversity of programming mistakes.

Possible directions for future research include identifying ways to use MRs to perform automatic validation to detect security issues that concern users, and exploring how users can employ MRs to specify their security requirements. One of the greatest advantages of MRs is that, once they are identified, testing can be fully automated.

Additional research is also needed to develop new test quality and adequacy criteria involving MRs in the context of security testing, which can complement existing criteria. We anticipate that diversity will be an underlying principle—from the selection of testing and analysis methods, to the formulation of test-case generation strategies and results-verification approaches, all the way to the establishment of quality standards.

References

1. A. Vassilev and C. Celi, “Avoiding Cyberspace Catastrophes through Smarter Testing,” *Computer*, vol. 47, no. 10, 2014, pp. 102–106.
2. E.T. Barr et al., “The Oracle Problem in Software Testing: A Survey,” *IEEE Trans. Software Eng.*, vol. 41, no. 5, 2015, pp. 507–525.
4 V. Okun and E. Fong, “Fuzz Testing for Software Assurance,” *CrossTalk–J. Defense Software Eng.*, vol. 28, no. 2, 2015, pp. 35–37.
9. Z.Q. Zhou, S. Xiang, and T.Y. Chen, “Metamorphic Testing for Software Quality Assessment: A Study of Search Engines,” *IEEE Trans. Software Eng.*, vol. 42, no. 3, 2016, pp. 264–284.

www.cs.ecu.edu/reu/reufiles/read/metamorphicTesting-16.pdf). .

8. H. Liu et al., "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?" *IEEE Trans. Software Eng.*, vol. 40, no. 1, pp. 4–22, 2014.
11. T.Y. Chen et al, "A Revisit of Three Studies Related to Random Testing," *Science China Information Sciences*, vol. 58, no. 5, 2015, pp. 052104:1–052104:9.
12. M. Lindvall et al., "Metamorphic Model-Based Testing Applied on NASA DAT–An Experience Report," *Proc. 37th IEEE Int'l Conf. Software Eng.* (ICSE 15), 2015, pp. 129–138.
3. C. Brubaker et al., "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations," *Proc. IEEE Symp. Security and Privacy*, 2014, pp. 114–129.
5. T.Y. Chen, T.H. Tse, and Z.Q. Zhou, "Semi-proving: An Integrated Method for Program Proving, Testing, and Debugging," *IEEE Trans. Software Eng.*, vol. 37, no. 1, 2011, pp. 109–125.
13. V. Le, M. Afshari, and Z. Su, "Compiler Validation via Equivalence Modulo Inputs," *Proc. 35th ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 14), 2014, pp. 216–226.
14. J. Regehr, "Finding Compiler Bugs by Removing Dead Code," 20 June 2014; http://blog.regehr.org/archives/1161.
15. M. Velez, "Finding and Understanding Bugs in Obfuscators, 2013; https://bitbucket.org/martinvelez/obfuscator_bugs_paper/downloads.
16. E. Selay, Z.Q. Zhou, and J. Zou, "Adaptive Random Testing for Image Comparison in Regression Web Testing," *Proc. IEEE Int'l Conf. Digital Image Computing: Techniques and Applications* (DICTA 14), 2014, pp. 1–7.
7. Z.Q. Zhou et al. "Automated Functional Testing of Online Search Services," *Software Testing, Verification and Reliability*, vol. 22, no. 4, 2012, pp. 221–243.
6. T.Y. Chen, T.H. Tse, and Z. Zhou, "Fault-Based Testing in the Absence of an Oracle," *Proc. 25th Ann. IEEE Int'l Computer Software and Applications Conf.* (COMPSAC 01), 2001, pp. 172–178.

Tsong Yueh Chen *is a professor of software engineering at Swinburne University of Technology, Australia. His main research interest is in software testing. Chen received his PhD from the University of Melbourne. He is a Senior Member of IEEE. Contact him at tychen@swin.edu.au.*

Fei-Ching Kuo *is a senior lecturer at Swinburne University of Technology, Australia. Her current research interests include software analysis, testing, debugging, and repair. Kuo received the PhD degree from Swinburne. She is a member of the IEEE. Contact her at dkuo@swin.edu.au.*

Wenjuan Ma *is an MPhil student in the School of Computing and Information Technology at the University of Wollongong (UoW). Her research interests include software testing and analysis. Ma received a BS in management from Beijing University of Chemical Technology. Contact her at wm230@uowmail.edu.au.*

Willy Susilo *is a professor in and head of the School of Computing and Information Technology at the UoW. His research interests include cryptography and network and cyber security. Susilo received a PhD in computer science with an emphasis on cryptography from UoW. He is a Senior Member of IEEE. Contact him at wsusilo@uow.edu.au.*

Dave Towey *is an assistant professor in the School of Computer Science at the University of Nottingham Ningbo China. His research interests include software testing, computer security, and technology-enhanced education. Towey received a PhD in computer science from the University of Hong Kong. He is a member of IEEE and ACM. Contact him at dave.towey@nottingham.edu.cn.*

Jeffrey Voas *is a computer scientist at the US National Institute of Standards and Technology. His research interests include the Internet of Things and fundamental computer science shortcomings. Voas received a PhD in computer science from the College of William and Mary. He is a contributing editor for* Computer*'s Security column and a Fellow of IEEE and the American Association for the Advancement of Science (AAAS). Contact him at j.voas@ieee.org.*

Zhi Quan Zhou *is a senior lecturer in software engineering at UoW. His research interests include software testing and debugging, security testing, and citation analysis. Zhou received a PhD in software engineering from the University of Hong Kong. Zhou is the corresponding author for this article. Contact him at zhiquan@uow.edu.au.*