1 THE TEST ORACLE PROBLEM

Internet and systems security are becoming a major concern, as deploying inadequately tested software can have serious consequences. To avoid “cyberspace catastrophes,” there is a need to adopt smarter software testing techniques [1], especially when testing security-related functionality and behavior.

Software testing, however, has a fundamental challenge: the oracle problem [2]. A test oracle is a mechanism against which testers can decide the correctness of test case execution outcomes. The majority of software testing techniques assume that an oracle is available and can be practically applied. In some circumstances, however, such an assumption does not hold true — a situation known as the oracle problem. To enhance cybersecurity by means of testing, the oracle problem must be addressed.

1.1 The Oracle Problem when Testing Security-related Software

When testing security-enhancing features of software, the oracle problem can become particularly severe. The use of different kinds of cryptographic algorithms and the complexity of the applications and their environments may mean that an oracle is often unavailable, or is theoretically available but practically is too expensive to be used.

Consider, for example, the testing of the certificate validation logic in SSL/TLS implementations [3]. If the program under test (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 Brubaker et al. [3] have pointed out, manually deciding test certificate validity does not scale; and automating this procedure “essentially requires re-implementing certificate validation, which is impractical and has high potential for bugs of its own.” However, because a number of independently implemented programs performing X.509 certificate validation exist (such as OpenSSL, NSS, and GnuTLS), Brubaker et al. were able to compare the outputs of these programs for the same input certificates, with discrepancies indicating that some implementations were incorrect.

In situations where multiple implementations of the same specification cannot be obtained, the oracle problem becomes more serious. Consider the testing of code obfuscators, which transform a program’s original code into an equivalent, less readable form, to prevent it from being analyzed and understood by attackers. Testing an obfuscator requires being able to determine if the input (original) code and the output (obfuscated) code are equivalent, which can be extremely difficult.

To the best of our knowledge, this is the first published report examining the testing of code obfuscators’ functional correctness, including the detection of real bugs (as will be presented in Section 3).

1.2 The Oracle Problem in Negative Testing

Two approaches to software testing are: positive testing, which uses valid input as test cases to ensure that a program behaves as expected in normal situations; and negative testing, which checks whether or not a program behaves reasonably when the input is invalid. Although positive testing is common to most software development, negative testing may often be omitted (perhaps due to resource constraints), potentially allowing security holes to persist into the released software [1], [4].

Fuzz testing, or fuzzing, is an important negative testing technique, where the PUT is tested using invalid, random, or semi-random inputs. Fuzzing is simple in concept, easy to implement, and supported by tools called...
fuzzers. Fuzzers may crash a system in unexpected ways, and thus may offer a high benefit-to-cost ratio. Fuzzing, therefore, is recognized as an efficient automatic testing technique for the detection of 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 semi-random input data is extremely difficult, if not impossible. To cope with this, fuzzing only looks for crashes, or some other undesirable behavior of the PUT. Because fuzzing does not check the correctness of each individual output, millions of test cases may be executed before finding a crash [4]. It should be noted that many bugs, such as logic errors [5], do not crash the PUT, but instead produce incorrect output, a type of failure which is much more difficult to detect than a system crash. The notorious Heartbleed bug [1], for example, does not cause a crash, and therefore cannot be detected by simple fuzzing [4].

In the rest of this article, we will address the above issues using the concept of metamorphic testing.

2 Metamorphic Testing (MT)

MT has been developed from a new perspective on testing: instead of focusing on the correctness of each individual output of a PUT, MT looks at the relationships among the inputs and outputs of multiple PUT executions. Such relationships are called metamorphic relations (MRs), and are necessary properties of the intended program's functionality. In MT, even if a test case does not reveal a failure, it can be used to generate follow-up test cases by referring to the selected MRs, and the PUT can then be further tested automatically, regardless of whether or not an oracle is available. In other words, MT alleviates the oracle problem in software testing and analysis [5]–[9].

Consider, for example, a PUT implementing the sine function. The property \( \sin(x) = \sin(180 - x) \) can be identified as an MR. If \( t = 32.875 \) is a source test case, and gives the output 0.543, a result which may not be easily verified due to the oracle problem, then (regardless of whether or not an oracle is available), MT can be used to suggest a follow-up test case \( t' = 180 - 32.875 \). If, after taking rounding errors into consideration, the two outputs are not equal, then a failure is revealed. MT has been applied to test various applications, ranging from numerical programs performing scientific computation to non-numerical programs such as search engines. Various MRs have been identified for those applications, which include, but are not limited to, identity relations.

MT has been proven to be highly effective in detecting failures [8], [10], [11], as illustrated by the following: In a PLDI'14 Distinguished Paper [12], researchers at UC Davis tested compilers “based on a particularly clever application of metamorphic testing” [13]. Basically, their MR is a special instance of the following: If source programs \( P \) and \( P' \) are equivalent on input \( I \), then their respective object programs \( O \) and \( O' \), generated by the compiler, should also be equivalent with respect to \( I \). In their study, \( P' \) is constructed as follows: First, execute \( O \) using arbitrary input \( I \), and record code coverage information with respect to \( P \). Then, create \( P' \) by randomly pruning some unexecuted “dead” statements from \( P \) as recorded in the first step. A compiler bug is reported if the output of \( O' \) on \( I \) has changed. Le et al. generated “147 confirmed, unique bug reports for GCC and LLVM alone,” of which “more than 100 have already been fixed” [12]. The detection of compiler bugs is particularly important as compilers are used to compile other programs, some of which may perform safety- or security-related functions.

3 Detecting Obfuscator Bugs using MT

As explained in Section 1.1, some security-related software can be difficult to test. In this section, we report on a case study of using MT to test code obfuscators.

3.1 A Need for Diverse Testing Techniques

Inspired by the success in compiler testing, UC Davis researchers also attempted to test C obfuscators by applying well-known compiler validation techniques [14]. In their testing process, each test case was a C source program \( P \), generated by CSmith, a random program generator. An obfuscator \( O \) produced an obfuscated source program \( O(P) \), and GCC was then used to compile \( P \) into executable code \( C(P) \), and \( O(P) \) into executable code \( C(O(P)) \). Finally, the tester ran both \( C(P) \) and \( C(O(P)) \), comparing their outputs: If the outputs did not match, then it would be concluded that the obfuscator was at fault (assuming that GCC was correct). For each of the two obfuscators under test, millions of different test cases have been run. However, the researchers report that they “have not discovered any bugs” and that “bugs are hard to find.”

This experience suggests that there is a need for more diverse testing techniques for obfuscators to be tested more effectively. In the following, we will address this challenge using MT: Instead of focusing on the correctness of each individual obfuscated program, we look at the relationships among multiple obfuscated programs — this is a perspective that has not been previously attempted.

3.2 Subject Programs

In this study, we tested four real-world obfuscators: Cobfusc (open source); Stunnix (commercial); Tigress (free software, but not open source), and Obfuscator-LLVM (open source).

Cobfusc (http://manpages.ubuntu.com/manpages/hardy/man1/cobfusc.1.html) is a Linux utility that makes a C source file unreadable, but compilable.

Stunnix is “a leader in providing advanced solutions for source code obfuscation” (http://stunnix.com). Its clients include many large organizations including the US Army and Fortune 500 companies. Stunnix supports...
several programming languages, but we tested its C/C++ obfuscator, CXX-Obfus, which was previously tested at UC Davis, where no failure was detected [14]. Clients of CXX-Obfus include Siemens, Ericsson, Sybase, Bosch, DELL, General Electric, Motorola, and Cryptography Research.

Tigress (http://tigress.cs.arizona.edu) is a C obfuscator that supports novel defenses against both static and dynamic reverse engineering.

LLVM (http://www.llvm.org/) is a well-known compiler infrastructure project, with users including Adobe Systems, Apple, Intel, and Sony. Obfuscator-LLVM (https://github.com/obfuscator-llvm/obfuscator/wiki) is a part of the LLVM compilation suite, used to provide increased software security. Given a C source program, Obfuscator-LLVM can be enabled by running LLVM’s front end compiler Clang with specific options. The output is an obfuscated and compiled binary code.

3.3 Identification of Diverse MRs

MT tests programs by referring to predefined MRs. Because programmers can make various mistakes, we believe that diverse MRs, when used together, would have a stronger fault-detection capability than a single MR. We therefore identified the following MRs:

(1) MR1: The first MR states that, if two different source programs \( P_1 \) and \( P_2 \) are functionally equivalent, then their obfuscated versions \( O(P_1) \) and \( O(P_2) \) are also functionally equivalent and, therefore, the compiled obfuscated executable programs \( C(O(P_1)) \) and \( C(O(P_2)) \) should have equivalent behavior — in the sense that they should give the same outputs for the same inputs.

To conduct testing based on this MR requires generation of equivalent source programs \( P_1 \) and \( P_2 \). Two automatic approaches for this involve either (i) use of a separate tool (such as a script written by the tester); or (ii) use of the obfuscator itself. We use MR1.1 and MR1.2 to denote these two approaches, respectively. An example of MR1.1 is: \( P_1 \) is constructed as 
\[
\text{If (condition) \{ do A \} else \{ do B \}}
\]

whereas \( P_2 \) is constructed as 
\[
\text{If (not(condition)) \{ do B \} else \{ do A \}}
\].

An example of MR1.2 is: \( P_1 \) is an arbitrary program and \( P_2 \) is \( O(P_1) \) — we use the obfuscator on \( P_1 \) to generate the (supposedly) equivalent program \( O(P_1) \).

This process can be repeated, and in the experiment we ran the obfuscation twice to obtain \( P_2 = O(O(P_1)) \).

(2) MR2: The second MR states that an obfuscator should generate behaviorally equivalent programs for the same input program, regardless of the environment the obfuscator is run under. In our experiment, we considered a specific kind of "environment," namely, time, with MR2 re-stated as: For the same input program, the obfuscator should generate behaviorally equivalent programs regardless of when the obfuscator is run.

1. It is possible that \( O(P_1) \) is not equivalent to \( P_1 \), when the obfuscator is faulty, in which situation MR1.2 can still be applied to test the obfuscator and reveal the fault.

\[
\begin{align*}
\text{if (i > j)} & \quad \text{if (i > (long)j + 116) - 116) } \\
& \quad \{ \text{i = (i + (10L + 116)) - 116; } \\
& \quad \text{else } \\
& \quad \text{i = (i + (10L + 116)) - 116; } \\
\end{align*}
\]

\[
\begin{align*}
\text{if (i <= j)} & \quad \text{if ((int)((i > (long)j + 116) - 116)) } \\
& \quad \{ \text{i = (i - (10L + 116)) + 116; } \\
& \quad \text{else } \\
& \quad \text{i = (i - (10L + 116)) + 116; } \\
\end{align*}
\]

Fig. 1: Tigress failure (version: Linux x86-64-unstable revision 1676), detected against MR1.1.

(3) MR3: The final MR differs from the other three in that, whereas MR1.1, MR1.2, and MR2 examine the behavioral equivalence of the compiled obfuscated programs by running them on the same sets of inputs, MR3 only looks at the obfuscated source codes, without compiling them. MR3 checks whether the obfuscation rules have been applied consistently every time the obfuscator is run. Although it is assumed that the tester has no detailed knowledge of the obfuscation rules adopted by the obfuscators, the tester can still check whether the outputs are consistent. For example, if a variable name in program \( P \) was obfuscated when the obfuscator was run yesterday, then the same variable name should still become obfuscated when the obfuscator is run today.

Because Obfuscator-LLVM only generates obfuscated binary code, without showing the obfuscated source code, MR1.2 and MR3 are not applicable to it.

3.4 Issues detected

We tested the subject obfuscators using 500 randomly generated source test cases (C programs), finding bugs or other issues in every obfuscator under test. We also observed different MRs detecting different types of issues. We next present one detected issue for each MR.

(1) MR1.1: Figure 1 shows excerpts of input files that revealed a failure in Tigress, when tested against MR1.1. Program \( P_1 \) (the source test case) has two integer variables \( i \) and \( j \), each of which is assigned an initial value. Then an \( if \) statement is executed: If \( i > j \) then \( i \) is set to \( i - 10 \), otherwise \( i \) is set to \( i + 10 \). Finally, the value of \( i \) is printed. The upper left part 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 part of Figure 1. \( O(P_1) \) and \( O(P_2) \) are the obfuscated codes of \( P_1 \) and \( P_2 \), the essential parts of which are shown in the upper and lower right parts of Figure 1, respectively. 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: a bug in Tigress was therefore detected.

In the case above, Tigress incorrectly obfuscated the \( P_1 \) statement "if \((i > j)\)" into the \( O(P_1) \) statement "if
Fig. 2: Obfuscator-LLVM (version 3.4) command line.

\[(int\ )((i > (long\ )j + 116) - 116))\] In the C language, the expression \(i > (long\ )j + 116\) is evaluated to either true ("1") or false ("0"). Therefore, the expression \((i > (long\ )j + 116) - 116\) is evaluated to either "-115" or "-116," both of which are non-zero and hence mean true. This means that the if statement of \(O(P_1)\) will always take the true branch, and the false branch is unreachable. This program is therefore not equivalent to \(P_1\). Similarly, the false branch of \(O(P_2)\) is also unreachable. When testing against \(MR_{1.1}\), \(C(O(P_1))\) and \(C(O(P_2))\) are run on the same input (e.g., \(i = j = 1000\)). After the if statement, \(C(O(P_1))\) sets \(i\) to \(i - 10\) (e.g., \(990\)), but \(C(O(P_2))\) sets it to \(i + 10\) (e.g., \(1010\)), thus generating different outputs. A bug in the obfuscator was therefore revealed.

One may argue that the bug could also be detected without MT, by compiling \(P_1\) into \(C(P_1)\), running \(C(P_1)\) and \(C(O(P_1))\) on the same input, and then comparing their outputs. It should be noted, however, that this conventional testing method can detect a failure only when the if statement of \(P_1\) takes the false branch (that is, only when the initial value of \(i\) is less than or equal to the initial value of \(j\)). In contrast, MT guarantees the detection of the bug, regardless of the initial values of \(i\) and \(j\), and therefore appears superior to conventional testing methods, again emphasizing that testing should be conducted from diverse perspectives.

(2) \(MR_{1.2}\): A failure of Cobfusc (in the package cutils version 1.6) was detected by \(MR_{1.2}\) as follows. A source test case \(P_1\) included the following statement:

\[
\text{int } k = 20; \ //Rz5Wq3OCvuqsA30uaEY0Evc95AI
\]

We then recursively called the obfuscator, Cobfusc, to construct \(P_2\) as \(O(O(P_1))\). \(O(P_1)\) and \(O(O(P_1))\) were expected to be equivalent but, to our surprise, \(O(O(P_1))\) could not pass the compiler because it included the following two lines of code (with a syntactically incorrect second line):

\[
\text{int } k = (5^5*111^0+0)*17276; \ //Rz5Wq3OCvuqsA30uaEY0Evc95AI
\]

The obfuscator had incorrectly moved the comment "Rz5Wq3OCvuqsA30uaEY0Evc95AI" from its original line, into a separate new line without the "//", causing a compiler error. A bug in Cobfusc was therefore detected.

(3) \(MR_2\): \(MR_2\) states that when an obfuscator is run at different times for the same program, the output programs should be equivalent. Figure 2 (left), line 1, shows a source program \(PBP_c\) compiled by Clang — the command line parameters enabled the use of Obfuscator-LLVM. The compiled obfuscated executable program (a.out) was run in line 2 with an input of 10000022, producing the output 10000022 (line 3). Lines 4, 5, and 6 repeat the above obfuscation-compilation-execution procedure, but this time producing different output (14195494). Lines 7, 8, and 9 repeat the procedure once more, producing the output 10000022. It can be concluded, therefore, that the three obfuscated executable programs are not equivalent.

The issue was caused by the following statements in \(PBP_c:\)

\[
\text{int } i; \text{ int } j; i=atoi(argv[1]); i=i+j; \text{ printf("%d\n",i);} \;
\]

In this piece of code, \(i\) is initialized with the input value (10000022 in the above executions) and then updated by the statement "i=i+j;" — here, \(j\) is used without initialization and, therefore, the value of \(i\) after this statement cannot be predicted. This value of \(i\) is printed by the last printf statement. Figure 2 (left) 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 equivalent. To further investigate this issue, the same \(PBP_c\) program was again compiled using Clang, but without enabling the obfuscation function, as shown in Figure 2 (right). It was observed that the compiled executable programs (a.out) consistently produced the same output (14195494), regardless of the number of times the compiler was run.

In summary, Cobfusc generated behaviorally equivalent binary code (a.out) whenever it was compiled using \(PBP_c\) with obfuscation disabled (Figure 2 (right)), but this was not the case when obfuscation was enabled (Figure 2 (left)).

(4) \(MR_3\): \(MR_3\) involves checking that obfuscated source files based on the same input source file are consistent. Figure 3 (left) shows an excerpt of source code before obfuscation. We ran Stunnix twice on this source code to generate two obfuscated output files, excerpts of which are shown in Figure 3 (middle) and (right). A comparison of these two output files reveals the following inconsistency: Line 6 of Figure 3 (middle) is the same as the original code, but line 6 of Figure 3 (right) is obfuscated. Although the obfuscation rules of Stunnix are unknown, this inconsistent behavior is undesirable: If a source statement is obfuscated in one run, it should also be obfuscated in other runs, allowing any confidential information to always be protected. An issue with Stunnix was thus revealed by \(MR_3\).

3.5 Summary

Obfuscators are important tools to help protect confidential software elements. As reported by Velez [14], however, it is difficult to find obfuscator bugs, even with...
advanced compiler testing techniques. In this section, we reported on obfuscator testing using MT. We identified a small number of quite diverse MRs and, with a small set of test cases, tested four well-known, real-world obfuscators, detecting issues in every one of them. This study is further evidence supporting our argument that testing should be conducted from diverse perspectives.

4  DETECTING WEB FAILURES USING MT

We applied MT to test the Internet banking login page of the National Australia Bank (NAB, https://www.nab.com.au), looking for compatibility issues between the website and the client side. From the perspective of a website user (not developer), we designed a simple MR: When different users log into a (local) computer using different user names, they should always be able to follow the same steps to navigate to the website and click on the Internet banking “Login” button. In other words, changing of user names on a local computer should not affect access to the Internet banking, as long as all the user accounts’ local settings are standard. During testing, screenshots of different user sessions were also automatically captured and compared to identify potential issues in the website GUI (readers interested in Web testing using screenshot comparison are referred to Selay et al. [15]).

Figure 4 shows an Internet banking login failure, automatically detected by our test driver. The left part of the figure corresponds to where a tester logged into 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. The right part of Figure 4 shows what happened when the tester logged into the same computer with a different user name, using the same browser to open the same website (the environment settings were all standard): The website GUI was not displayed correctly, and neither the red “Login” button, nor the grey “Login” link could be clicked.

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

5  MT FOR NEGATIVE TESTING

This section discusses how the new perspectives from MT can be used to guide negative testing.

5.1  Heartbleed: A “Pattern” of Mistakes

The Heartbleed bug is probably the most widely known cybersecurity breach in recent years [1]. The bug was 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 length of payload must be payload_length. When implementing this, the programmer assumed that the relationship between payload and payload_length would always hold true and, therefore did not include any bounds checking code. Furthermore, although the implementation was reviewed by a core OpenSSL developer, the reviewer had the same wrong assumption and did not find the bug. In other words, the bug is a result of a common type of mistake: overlooking the possibility of some parameters taking a value outside the expected range.

5.2  MT Can Alleviate the Problem

The Heartbleed bug cannot be detected by simple fuzzing because it only produces incorrect output, and does not crash the system. It is therefore necessary to incorporate multiple tools or techniques to increase fault detection capability [1]. Compared with most other testing and analysis methods, MT has a different perspective: It considers the relationship among multiple executions of the PUT. Consider the HeartbeatMessage shown in Figure 5. To conduct MT, the tester will ask the question: “What if I change some of the parameter values?” More specifically, consider a source test case \( t = (\text{type}_1, \text{length}_1, \text{payload}_1, \text{padding}_1) \), where \( \text{type}_1, \text{length}_1, \text{payload}_1, \) and \( \text{padding}_1 \) represent concrete values of the parameters. To identify an MR, the tester will ask the following questions:

1. “What if I change \( \text{type}_1 \) to a different value?”
2. “What if I change \( \text{length}_1 \) to a different value?”
3. “What if I change \( \text{payload}_1 \) to a different value?”
4. “What if I change \( \text{padding}_1 \) to a different value?”

Fig. 4: MT detected an Internet banking login failure (https://www.nab.com.au) using an Acer Chromebook (model no. Q1VZC). Left: Normal with local guest account. Right: Failure with any non-guest local account.

Fig. 5: Excerpt from Section 4 of RFC 6520 (available: http://tools.ietf.org/html/rfc6520).
5. “What if I change two or more parameters?”

Asking these questions will lead the tester to think beyond the “normal” range of values or value combinations of the parameters. In other words, MT can lead the tester to think about negative testing. In this example, as soon as question 2 is asked, the tester will immediately realize that payload_length may not equal the actual length of payload, and hence may construct a follow-up test case ‘t’ by increasing the value of length, while keeping the other parameters unchanged. The MR will require that the program’s outputs for t and t’ be different (as per RFC 6520): It should return a normal message for t, but “silently” discard t’. When the faulty program is tested against this MR, a failure will be revealed.

It should be noted that, in the above example, there is indeed an oracle (that is, the RFC 6520), but MT still has the advantage of leading to negative testing. The Heartbleed bug could also be detected by using a fuzzer in conjunction with some dynamic analysis tools — tools which perform run-time monitoring for predefined types of memory errors [1]. Compared with dynamic analysis, MT is not limited to predefined error patterns — it can detect both system crashes (in which case the MRs are violated) and other types of errors.

As Zhou et al. have shown [7], MT can be combined with fuzzing: When testing Microsoft Live Search, a random string “GLIF OR 5Y4W” 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 — 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.

6 Conclusion and Future Work

In this article, we have shown that MT can help to achieve negative testing as well as to alleviate the oracle problem when testing security-related functionality and behavior. We have shown that MT successfully revealed real-life bugs not previously detected by other testing methods. This is not only because MT can be performed in the absence of an oracle, but also because it is based on a perspective not previously used by conventional testing techniques. As recently emphasized by Chen et al. [10], this is not to say that MT is necessarily better than the other testing methods, but rather that testing should be conducted from diverse perspectives, because programmers can make various kinds of mistakes.

A future research direction is to study how to use MRs to perform automatic validation to detect security issues that concern users, and how users can specify their security requirements using MRs. One of the greatest advantages of using MRs is that, once identified, the testing can be fully automated.

Further research is also needed to develop new test quality and adequacy criteria involving MRs, in the context of security testing. Such criteria will be complementary to the existing ones. We anticipate that the concept of diversity will be an underlying principle, from the selection of testing and analysis methods to test case generation strategies, through to result verification approaches, and to quality standards, and more.

REFERENCES