Test Case-Aware Combinatorial Interaction Testing

Cemal Yilmaz

Abstract—The configuration spaces of modern software systems are too large to test exhaustively. Combinatorial interaction testing (CIT) approaches, such as covering arrays, systematically sample the configuration space and test only the selected configurations by using a battery of test cases. Traditional covering arrays, while taking system-wide inter-option constraints into account, do not provide a systematic way of handling test case-specific inter-option constraints. The basic justification for t-way covering arrays is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. In this work we hypothesize, however, that in the presence of test case-specific inter-option constraints, many such behaviors may not be tested due to masking effects caused by the overlooked test case-specific constraints. For example, if a test case refuses to run in a configuration due to an unsatisfied test case-specific constraint, none of the valid option setting combinations appearing in the configuration will be tested by that test case. To account for test case-specific constraints, we introduce a new combinatorial object, called a *test case-aware covering array*. A t-way test case-aware covering array is not just a set of configurations as is the case in traditional covering arrays, but a set of configurations, each of which is associated with a set of test cases, such that all test case-specific constraints are satisfied and that, for each test case, each valid combination of option settings for every combination of t options appears at least once in the set of configurations that the test case is associated with. We furthermore present three algorithms to compute test case-aware covering arrays. Two of the algorithms aim to minimize the number of configurations required (one is fast, but produces larger arrays, the other is slower, but produces smaller arrays), whereas the remaining algorithm aims to minimize the number of test runs required. The results of our empirical studies conducted on two widely-used highly-configurable software systems suggest that test case-specific constraints do exist in practice, that traditional covering arrays suffer from masking effects caused by the ignorance such constraints, and that test case-aware covering arrays are better than other approaches in handling test case-specific constraints, thus avoiding masking effects.

Index Terms—Software quality assurance, combinatorial interaction testing, covering arrays.

1 Introduction

General-purpose, one-size-fits-all software solutions are not acceptable in many application domains. For example, web servers (e.g., Apache), databases (e.g., MySQL), and application servers (e.g., Tomcat) are required to be customizable to adapt to particular runtime contexts and application scenarios. One way to support software customization is to provide configuration options through which the behavior of the system can be controlled.

While having a configurable system promotes customization, it creates many system configurations, each of which may need extensive QA to validate. Since the number of configurations grows exponentially with the number of configuration options, exhaustive testing of all configurations, if feasible at all, does not scale well.

Combinatorial interaction testing (CIT) approaches systematically sample the configuration space and test only the selected configurations [3], [9], [14], [22], [34]. These approaches take as input a configuration space model. The model includes a set of configuration options, their possible settings, and a set of system-wide inter-option constraints that explicitly or implicitly invalidate some configurations, i.e., not all configurations may be valid. Given a configuration space model, CIT approaches compute a small set of valid configurations, called a *t-way covering array*, in which each valid combination of option settings for every combination of t options appears at least once [9].

Once a covering array is generated, the system is then tested by running its test cases in all the selected configurations. By doing so, traditional covering arrays assume that all test cases can run in all the selected configurations. In this work we argue that the test cases of configurable systems are likely to have assumptions about the underlying configurations. That is, not all test cases may run in all configurations even if those configurations satisfy the system-wide constraints. For example, in a study conducted on Apache, a highly-configurable HTTP server, and MySQL, a highly-configurable database management system, we observed that 378 out of 3789 Apache test cases and 337 out of 738 MySQL test cases had test-case specific inter-option constraints. While the system-wide constraints determined the set of valid ways the system under test could be configured, a test case-specific constraint determined the

• *C. Yilmaz is with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey.*
E-mail: see http://people.sabanciuniv.edu/cyilmaz

o1	o2	o3	o4	$t1$	$t2$	$t3$
1	1	1	1	S	P	P
1	1	0	0	S	P	P
1	0	1	0	S	P	P
1	0	0	1	S	P	P
0	1	1	0	P	S	P
0	1	0	1	P	S	P
0	0	1	1	P	S	P
0	0	0	0	P	S	P

(a)

o1	o2	o3	o4	tests	o1	o2	o3	o4	tests
0	1	1	1	$\{t1\}$	1	1	1	1	$\{t2, t3\}$
0	1	0	0	$\{t1\}$	1	1	0	0	$\{t2, t3\}$
0	0	1	0	$\{t1\}$	1	0	1	0	$\{t2, t3\}$
0	0	0	1	$\{t1\}$	1	0	0	1	$\{t2, t3\}$
0	1	1	0	$\{t1, t3\}$	1	1	1	0	$\{t2\}$
0	1	0	1	$\{t1, t3\}$	1	1	0	1	$\{t2\}$
0	0	1	1	$\{t1, t3\}$	1	0	1	1	$\{t2\}$
0	0	0	0	$\{t1, t3\}$	1	0	0	0	$\{t2\}$

(b)

Fig. 1: A traditional 3-way covering array (a) vs. a 3-way test case-aware covering array (b).

set of configurations in which the respective test case could run. These test case-specific constraints were encoded in the test oracles of our subject applications, indicating that they were known to the developers. When a test case-specific constraint of a test case was not satisfied by a configuration, the test case simply *skipped* the configuration, i.e., the test case refused to run in the configuration. For instance, a number of Apache test cases were specifically designed to test the Distributed Authoring and Versioning (DAV) feature of the Apache server – a feature that needs to be explicitly configured into the system. Therefore, these test cases assumed that the server was already configured with the DAV feature. If not, they simply refused to run. Other test cases in the test suite were unaffected by this aspect of server configuration and ran regardless of the setting of this configuration option.

It is important to note that running configuration-dependent test cases in an ad hoc manner only in configurations supporting the required features, such as DAV, is not necessarily sufficient, as the interactions between the required features and the rest of the configurable features may still need to be tested. Therefore, we still need a systematic way of testing the interactions, such as the one provided by covering arrays.

On the other hand, traditional covering arrays, while handling system-wide constraints, do not provide a systematic way of handling test case-specific constraints. Assuming the existence of a well constructed test suite, the basic justification for traditional t-way covering arrays is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. We hypothesize however that, in the presence of test case-specific constraints, many such behaviors are not actually tested because of masking effects [15] caused by the overlooked test case-specific constraints. For instance, when a test case skips a configuration due to an unsatisfied test case-specific constraint, none of the option setting combinations appearing in the configuration will be tested by that test case.

Figure 1a illustrates masking effects in a hypothetical scenario. In this scenario, the system under test has 4 configuration options ($o1$, $o2$, $o3$, and $o4$), each of which takes a boolean value (0 or 1). The test suite contains 3 test cases ($t1$, $t2$, and $t3$). There are no system-wide constraints. However, test cases $t1$ and $t2$ have some test case-specific constraints: $t1$ can run only in configurations in which $o1$=0, and $t2$ can run only in configurations in which $o1$=1. Test case $t3$, on the other hand, has no test case-specific constraints. In our hypothetical scenario, by mimicking the way traditional covering arrays are used in practice, a 3-way covering array is created and then all the test cases are executed in all the selected configurations (Figure 1a). Literals P and S indicate a test success and a test skip, respectively.

There are 20 valid 3-way combinations of option settings to be tested by each of $t1$ and $t2$, and 32 valid combinations for $t3$. Consider the test case $t1$. Since $t1$ skipped the first 4 configurations, the 3-way option setting combinations for options $o2$, $o3$, and $o4$ that appear in the first 4 configurations, were actually not tested by $t1$. As these 4 combinations appear nowhere else in the covering array, $t1$ never had a chance to test them. Similarly, $t2$ never had a chance to test the 4 valid 3-way combinations, which happened to appear only in the last 4 configurations skipped by $t2$. As a result, 8 out of 72 (11%) valid 3-way option setting combination-test case pairs were not tested at all, i.e., masked, due to the test skips caused by the overlooked test case-specific constraints.

In this work, to account for test case-specific constraints and avoid the harmful consequences of masking effects caused by them, we introduce a new combinatorial object, called a *test case-aware covering array*. Test case-aware covering arrays take as input a traditional configuration space model augmented with a set of test cases, each of which can have a test case-specific constraint. Given a configuration space model, a t-way test case-aware covering array is not just a set of configurations as is the case in traditional covering arrays, but a set of configurations, each of which is associated with a set of test cases, indicating the test cases scheduled to be executed in the configuration.

Figure 1b, as an example, presents a 3-way test case-aware covering array constructed for our hypothetical scenario. A t-way test case-aware covering array has the following properties: 1) For each test case, every valid t-way combination of option settings occurs at least once in the set of configurations in which the test case is scheduled to be executed and 2) No test case is scheduled to be executed in a configuration which violates the test case-specific constraints of the test case, or the system-wide constraints.

We furthermore present three algorithms to compute test case-aware covering arrays. Two of the algorithms aim to minimize the number of configurations required (one is fast, but produces larger arrays, the other is slower, but produces smaller arrays), whereas the remaining algorithm aims to minimize the number of test runs required. The results of our empirical studies conducted on two widely-used highly-configurable software systems, namely Apache and MySQL, strongly suggest that 1) test case-specific inter-option constraints do exist in practice, 2) traditional covering arrays suffer from masking effects caused by the ignorance of such constraints, and 3) test case-aware covering arrays are better than other approaches in handling test case-specific constraints, thus avoiding masking effects.

The remainder of this paper is organized as follows: Section 2 provides background information; Section 3 introduces test case-aware covering arrays; Section 4 presents three algorithms to compute them; Section 5 describes the empirical studies; Section 6 discusses the potential external threats to validity; Section 7 discusses the related work; and Section 8 presents concluding remarks and possible directions for future work.

2 Background Information

In this section we provide background information on traditional covering arrays and masking effects.

2.1 Traditional Covering Arrays

CIT approaches take as input a *configuration space model* $M{=}{<}O, V, Q{>}$. The model includes a set of configuration options $O{=}\{o_1, o_2, \ldots, o_n\}$, their possible settings $V{=}\{V_1, V_2, \ldots, V_n\}$, and a system-wide inter-option constraint Q (if any). In effect, the configuration space model implicitly defines a valid configuration space for testing.

Each option o_i ($1 \leq i \leq n$) in the configuration space model takes a value from a finite set of k_i distinct values $V_i{=}\{v_1, v_2, \ldots, v_{k_i}\}$ ($k_i{=}|V_i|$). Let s_{ij} be an option-value tuple of the form ${<}o_i, v_j{>}$, indicating that option o_i is set to value $v_j \in V_i$. Furthermore, let S_i be the set of all possible option-value tuples for option o_i, i.e., $S_i{=}\{{<}o_i, v_j{>}\text{: } v_j \in V_i\}$.

Definition 1. *A t-tuple* $\phi_t{=}\{s_{i_1j_1}, s_{i_2j_2}, \ldots, s_{i_tj_t}\}$ *is a set of option-value tuples for a combination of* t *distinct options, such that* $1 \leq t \leq n$, $1 \leq i_1 < i_2 < \ldots < i_t \leq n$, *and* $s_{i_pj_p} \in S_{i_p}$ *for* $p{=}1, 2, \ldots, t$.

Let $\hat{\Phi}_t$ be the set of all t-tuples for some $1{\leq}t{\leq}\, n$. Not all the t-tuples in $\hat{\Phi}_t$ may be valid due to the system-wide constraint Q. Assume a deterministic function $valid(\phi_t, Q)$, such that $valid(\phi_t, Q)$ is true, if and only if, ϕ_t is a valid t-tuple under constraint Q. Otherwise, $valid(\phi_t, Q)$ is false. The set of all *valid* t-tuples Φ_t under constraint Q is then defined as: $\Phi_t{=}\{\phi_t : \phi_t \in \hat{\Phi}_t \wedge valid(\phi_t, Q)\}$.

Definition 2. *Given a configuration space model* $M{=}{<}O, V, Q{>}$*, a valid configuration* c *is a valid n-tuple, i.e.,* $c \in \Phi_n$*, where* $n = |O|$.

Note that, in a valid configuration, each option defined in the configuration space model takes a valid value and the configuration (i.e., n-tuple) does not violate Q.

Definition 3. *Given a configuration space model* $M{=}{<}O, V, Q{>}$*, the valid configuration space* C *is the set of all valid configurations, i.e.,* $C{=}\{c : c \in \Phi_n\}$.

CIT approaches systematically sample the valid configuration space and test only the selected configurations. The sampling is carried out by computing a t-way covering array [9], where t is often referred to as the *strength* of the covering array.

Definition 4. *A t-way covering array* $CA(t, M{=}{<}O, V, Q{>})$ *is a set of valid configurations in which each valid t-tuple appears at least once, i.e.,* $CA(t, M{=}{<}O, V, Q{>}){=}\{c_1, c_2, \ldots, c_N\}$*, such that* $\forall\, \phi_t \in \Phi_t\, \exists\, c_i \supseteq \phi_t$*, where* $c_i \in C$ *for* $i{=}1, 2, \ldots, N$.

Once a covering array is computed, the system under test is validated by running its test suite in all the selected configurations. Since the amount of resources required for testing is a function of the covering array size (i.e., N), covering arrays are constructed so that all valid t-tuples are covered in a "minimum" number of configurations.

Furthermore, it is also of practical interest to guarantee the inclusion of certain configurations in covering arrays for testing. A widely-used mechanism for this purpose is the *seeding* mechanism [9], [13], [18]. Given a seed, which is typically a set of configurations, the covering array is constructed around the seed. Conceptually, all the t-tuples included in the seed are considered to be already covered and a new set of configurations are generated only to cover the rest of the valid t-tuples.

2.2 Masking Effects

We first introduced the concept of masking effects in one of our earlier works [15].

Definition 5. *A masking effect is an effect that prevents a test case from testing all valid combinations of option* *settings appearing in a configuration, which the test case is normally expected to test.*

A harmful consequence of masking effects is that they cause developers to develop false confidence in their test processes, believing them to have tested certain option setting combinations, when they in fact have not. One simple example of a masking effect (besides the ones caused by overlooked test case-specific constraints) would be an error that crashes a program early in the program's execution. The crash then prevents some configuration dependent behaviors, that would normally occur later in the program's execution, from being exercised. Unless the combinations controlling those behaviors are tested in a different configuration, or unless the error is fixed and the faulty configuration is re-tested, we cannot conclude that those configuration dependent behaviors have been tested.

Masking effects can be caused by many factors. System failures, unaccounted control dependencies among configuration options (i.e., option setting combinations that effectively cancel other option setting combinations), and incomplete or incorrect inter-option constraints can all perturb program executions in ways that prevent other configuration dependent behaviors from being tested.

In our previous work [15], we focused on masking effects caused by system failures and developed a feedback driven adaptive combinatorial testing approach to reduce their harmful consequences. At each iteration of the aforementioned approach, we detect potential masking effects, isolate their likely causes, and then schedule the t-tuples that are being masked, for testing in the subsequent iteration. The iterations end when, for each test case, all valid t-tuples have been "tested".

This work differs from our previous work in that we in this work address masking effects caused by overlooked test case-specific constraints rather than the ones caused by system failures, and that, as these constraints are known a priori, we compute static interaction test suites rather than the dynamic ones produced by the previous approach.

3 Test Case-Aware Covering Arrays

In this work we consider an inter-option constraint to be a constraint among option settings, which explicitly or implicitly invalidates some combinations of option settings. System-wide inter-option constraints apply to all test cases and define the set of valid ways the system under test can be configured. A test case-specific constraint, on the other hand, applies only to the test case that it is associated with and determines the configurations in which the test case can run.

It is important to note that expressing test case-specific constraints as system-wide constraints and then generating traditional covering arrays, is not an adequate solution for handling test case-specific constraints. One reason is that constraints for different test cases may conflict with each other, in which case no solution will be found. For example, in our hypothetical scenario discussed in Section 1, the constraints of $t1$ and $t2$ conflict; $t1$ cannot run when the binary option $o1$ has one setting and $t2$ cannot run when the same option has the other setting. Globally enforcing these conflicting constraints will not generate any covering arrays. Another reason is that, even if the test case-specific constraints do not conflict, enforcing them on all test cases can prevent the test cases from exercising some valid option setting combinations that are invalidated by other test cases. For example, enforcing the constraint of $t1$ on $t3$ prevents $t3$ from testing any combinations with $o1{=}1$, which are valid for $t3$.

In this work, to account for test case specific-constraints, we introduce test case-aware covering arrays. As is the case with traditional covering arrays, test case-aware covering arrays take as input a configuration space model M. The model contains a set of configuration options $O{=}\{o_1, \ldots, o_n\}$, their settings $V{=}\{V_1, \ldots, V_n\}$, and a system-wide inter-option constraint Q_s. Unlike traditional covering arrays, the configuration space model of test case-aware covering arrays additionally includes a set of test cases $\mathrm{T}{=}\{\tau_1, \tau_2, \ldots\}$. Each test case $\tau \in \mathrm{T}$, in addition to implicitly inheriting the system-wide constraint Q_s, can have a test case-specific constraint Q_τ. In the remainder of the paper, the collection of all test case-specific constraints is referred to as Q_T.

In the presence of test case-specific constraints, we define the set of valid t-tuples on a per-test case basis, since a valid t-tuple for a test case may be invalid for another test case. Let $\Phi_t^\tau{=}\{\phi_t : \phi_t \in \hat{\Phi}_t \wedge valid(\phi_t, Q_s \wedge Q_\tau)\}$ be the set of all valid t-tuples for a test case τ.

Definition 6. *A valid configuration c^τ for a test case $\tau \in \mathrm{T}$ is a valid n-tuple for τ, i.e., $c^\tau \in \Phi_n^\tau$, where $n = |O|$.*

Definition 7. *The valid configuration space C^τ for a test case $\tau \in \mathrm{T}$ is the set of all valid configurations for τ, i.e., $C^\tau{=}\{c : c \in \Phi_n^\tau\}$.*

Test case-aware covering arrays aim to ensure that each test case has a fair chance to test all of its valid t-tuples. To this end, each test case is scheduled to be executed only in configurations which are valid for the test case so that no masking effects can occur.

Definition 8. *A t-pair is a pair of the form $\lambda_t{=}{<}\phi_t, \tau{>}$, such that $\phi_t \in \Phi_t^\tau$ and $\tau \in \mathrm{T}$.*

Definition 9. *A t-way test case-aware covering array $TCA(t, M{=}{<}O, V, \mathrm{T}, Q_s, Q_\mathrm{T}{>}) = \{{<}c_1, \mathrm{T}_1{>}, \ldots, {<}c_N, \mathrm{T}_N{>}\}$ is a set of rows of the form ${<}c_i, \mathrm{T}_i{>}$, where $c_i \in C$ and $\mathrm{T}_i \subseteq \mathrm{T}$ for $i = 1, 2, \ldots, N$, such that each valid t-pair appears at least once, i.e., $\forall \tau \in \mathrm{T} \wedge \phi_t \in \Phi_t^\tau \; \exists {<}c_i, \mathrm{T}_i{>} : \phi_t \subseteq c_i \wedge \tau \in \mathrm{T}_i$ and $\tau \in \mathrm{T}_i \rightarrow c_i \in C^\tau$.*

In other words, for a given configuration space model, a t-way test case-aware covering array is a set of configurations, each of which is associated with a set of test cases, indicating the test cases scheduled to be executed in the configuration, such that 1) none of the selected configurations violate the system-wide constraint, 2) no test case is scheduled to be executed in a configuration that violates the test case-specific constraint of the test case, and 3) for each test case, every valid t-tuple appears at least once in the set of configurations in which the test case is scheduled to be executed. Figure 1b, as an example, presents a 3-way test case-aware covering array created for our hypothetical scenario depicted in Figure 1a. Since none of the test case-specific constraints are violated in this covering array, each test case has a chance to test all of its valid 3-tuples; no masking effects caused by test skips can occur.

Compared to traditional t-way covering arrays, handling test case-specific constraints is likely to increase the number of configurations required, as the t-tuples being masked in traditional arrays may need to be covered in additional configurations. However, this does not necessarily imply an increase in the number of test runs required, as the test cases are executed only in configurations that contribute to the coverage. For example, comparing the 3-way test case-aware covering array in Figure 1b to the traditional 3-way covering array in Figure 1a, we observe that, while the number of configurations doubles, the number of test runs stays the same as each array requires a total of 24 test runs. In this work, an execution of a test case in a configuration is considered to be a test run.

Therefore, when the goal is to test all valid t-pairs, then traditional t-way covering arrays will not guarantee the coverage in the presence of test case-specific constraints, whereas t-way test case-aware covering arrays, while guaranteeing a full coverage, will do so at the possible cost of increased number of configurations, but not necessarily increased number of test runs. For example, the 3-way test case-aware covering array in Figure 1b is optimal for our hypothetical scenario; no other arrays requiring fewer number of configurations or fewer number of test runs, exist. Therefore, testing all 3-pairs in this scenario is not possible without (at least) doubling the number of configurations. Thus, developers must decide between increasing the number of configurations to remove masking effects or accepting their harmful consequences. On the other hand, if the cost of configuring the system is negligible, then the 3-way test case-aware covering array in Figure 1b will remove all masking effects at no additional cost compared to the traditional 3-way covering array in Figure 1a; both arrays require the same number of test runs.

4 Computing Test Case-Aware Covering Arrays

In this section we present three algorithms to compute test case-aware covering arrays. A valuable observation we make is that there is often a trade-off between minimizing the number of configurations and minimizing the number of test runs in test case-aware covering arrays. An attempt to reduce one count often increases the other count. This trade-off plays an important role in minimizing the total cost of testing, especially when there is a profound practical difference between the cost of configuring the system and the cost of running the test cases.

The first two algorithms presented in this section, namely Algorithm 1 and Algorithm 2, assume a cost model in which the cost of running the test cases is negligible compared to that of configuring the system and the configuration cost is the same for all configurations. Thus, Algorithm 1 and Algorithm 2 aim to minimize the number of configurations required. On the other hand, the remaining algorithm, namely Algorithm 3, assumes an opposite cost model in which the cost of configuring the system is negligible compared to that of running the test cases and the execution cost is the same for all test runs. Thus, Algorithm 3 aims to minimize the number of test runs required.

4.1 Algorithm 1: Maintaining a separate configuration space model for each test case

A large number of algorithms have been proposed for constructing traditional covering arrays [26]. In our first algorithm, whose roots stem from one of our earlier works [15], we use traditional covering array construction as a computational primitive to generate test case-aware covering arrays. At a high level, we generate a separate traditional covering array for each test case, and, while doing so, we force the test cases to share configurations. The generated traditional covering arrays are then merged to construct the test case-aware covering array.

In particular, we assume a generator $\prod(t, M, S)$ that constructs a traditional t-way covering array around the seed S for the configuration space model M. Such covering array generators are commonplace [1].

Given a system-wide configuration space model $M{=}{<}O, V, \mathrm{T}, Q_s, Q_{\mathrm{T}}{>}$, for each test case $\tau \in \mathrm{T}$, we first create a separate configuration space submodel M_τ. The submodel M_τ, in addition to inheriting the configuration options O, their settings V, and the system-wide constraint Q_s from M, includes the test case-specific constraint of τ as a system-wide constraint. Since the submodels are maintained on a per-test case basis, conflicting test case-specific constraints do not pose any issues.

Once the configuration space submodels are created, we feed these models to Algorithm 1. The t-way

Algorithm 1 Computing a t-way test case-aware covering array Ψ_t^M by maintaining a separate configuration space submodel M_τ for each test case τ, where S_τ and $\Omega_t^{M_\tau}$ are the seed and the traditional t-way covering array created for τ, respectively.

Input M=<$O, V, \mathrm{T}, Q_s, Q_\mathrm{T}$>: Config. space model
Input t: Covering array strength

```
1: $\Psi_t^M \leftarrow \emptyset$
2: for each test case $\tau$ in $\mathrm{T}$ do
3:     $S_\tau \leftarrow \{c : c \in \Psi_t^M \wedge c \in C^\tau\}$
4:     $\Omega_t^{M_\tau} \leftarrow \prod(t, M_\tau, S_\tau)$
5:     $\Psi_t^M \leftarrow \Psi_t^M \bullet \Omega_t^{M_\tau}$
6: end for
7: return $\Psi_t^M$
```

test case-aware covering array Ψ_t^M to be computed, is initially empty (line 1). For each test case τ, we first compute a seed S_τ (line 3). The seed, out of all the configurations which have been so far included in the test case-aware covering array, contains only those configurations that do not violate the test case-specific constraints of τ. We then feed the seed S_τ and the configuration space submodel M_τ to the generator $\prod$. The output is a traditional t-way covering array $\Omega_t^{M_\tau}$ that covers all the valid t-tuples for the test case τ (line 4). The test case τ is then scheduled to be executed in all the selected configurations in $\Omega_t^{M_\tau}$, assuming that non-contributing configurations that might be present in the seed S_τ are eliminated by the generator.

We use the seeding mechanism to reduce the number of configurations required. Since $\prod$ marks all the t-tuples included in the seed as already covered and generates new configurations only to cover the rest of the valid t-tuples, having the seed forces the test cases to share configurations, i.e., test cases are scheduled to be executed in the same configurations as much as possible, reducing the total number of configurations required.

Once the traditional covering array for the test case τ is created, we merge it with the current test case-aware covering array Ψ_t^M (line 5). The merge operation is performed in such a way that test cases that are scheduled to be executed in the same configuration are collected in a set and then the newly constructed set is associated with the configuration. Finally, after processing all the test cases, we output the generated test case-aware covering array (line 7).

Figure 2 illustrates Algorithm 1 in an example. In the example, we have a configuration space model that contains 4 binary options, namely $o1$, $o2$, $o3$, and $o4$. The system is to be tested with 3 test cases: $t1, t2$, and $t3$. Test cases $t1$ and $t2$ do not have any test case-specific constraints, whereas $t3$ skips all configurations in which $o1{=}0 \wedge o4{=}0$ or $o2{=}0 \wedge o3{=}0$. There are no system wide constraints. Furthermore, the system is required to be tested with a 2-way test case-aware covering array.

To generate the test case-aware covering array, first, a separate configuration space submodel is created for each test case (Figure 2a-c). Then, a traditional 2-way covering array is created for each test case in the arbitrary order of $t1, t2$, and then $t3$. For $t1$, as no configurations have been scheduled for testing yet, an empty seed is used and a traditional 2-way covering array of size 5 is created (Figure 2a). The newly generated covering array is then used as the seed for $t2$. Since both $t1$ and $t2$ require to cover exactly the same set of 2-tuples, no additional configurations are needed for $t2$ (Figure 2b). In the figure, configurations that are reused from the previous iterations are marked with the character '*'. Next, out of the 5 configurations that have been so far scheduled for testing, the first 3 configurations, which do not violate the test case-specific constraint of $t3$ is used as the seed for $t3$. Figure 2c presents the traditional 2-way covering array computed for $t3$; all of the 3 configurations in the seed happen to be reused by $t3$. Finally, the traditional covering arrays are merged to generate the 2-way test case-aware covering array given in Figure 2d. The resulting array requires 8 configurations and 16 test runs.

4.2 Algorithm 2: Maintaining a single configuration space model

An attractive side of Algorithm 1 is that it can readily and quickly be implemented by using the existing covering array generators that support system-wide constraints and a seeding mechanism. One downside of the approach, though, is that the optimization is carried out on a per-test case basis. While the problem is being solved for a test case, the coverage requirements of the remaining test cases waiting to be processed are not taken into account. This leads to a loss of opportunity for further reducing the number of configurations. For example, changing the processing order of the test cases in Figure 2 can produce a smaller array.

In this section, to alleviate the shortcomings of Algorithm 1, we present a greedy algorithm that keeps a global view of all valid t-pairs to be covered. To this end, we maintain a single configuration space model for the system under test rather than a separate model for each test case.

Our algorithm operates in an iterative manner. At each iteration, we select the best configuration and the set of associated test cases, which cover the most number of previously uncovered t-pairs. The selection is then included in the test case-aware covering array and the t-pairs appearing in the selection are marked as covered. The iterations end when every valid t-pair is covered at least once.

Algorithm 2 presents the main loop of the proposed approach. We maintain a pool λ_t^T that keeps track of

Configuration Space Submodel for t1:

```
[options]
  o1,o2,o3,o4: {0, 1}

[constraints]
  #empty
```

Seed:
#empty

Covering Array for t1:

o1	o2	o3	o4
1	0	1	0
0	1	1	1
1	1	0	0
1	0	0	1
0	0	0	0

(a)

Configuration Space Submodel for t2:

```
[options]
  o1,o2,o3,o4: {0, 1}

[constraints]
  #empty
```

Seed:

o1	o2	o3	o4
1	0	1	0
0	1	1	1
1	1	0	0
1	0	0	1
0	0	0	0

Covering Array for t2:

o1	o2	o3	o4	
1	0	1	0	*
0	1	1	1	*
1	1	0	0	*
1	0	0	1	*
0	0	0	0	*

(b)

Configuration Space Submodel for t3:

```
[options]
  o1,o2,o3,o4: {0, 1}

[constraints]
  ¬(o1=0 ∧ o4=0)
    ∧¬(o2=0 ∧ o3=0)
```

Seed:

o1	o2	o3	o4
1	0	1	0
0	1	1	1
1	1	0	0

Covering Array for t3:

o1	o2	o3	o4	
1	0	1	0	*
0	1	1	1	*
1	1	0	0	*
0	1	0	1	
0	0	1	1	
1	1	1	1	

(c)

Test Case-Aware Covering Array

o1	o2	o3	o4	tests
1	0	1	0	{t1,t2,t3}
0	1	1	1	{t1,t2,t3}
1	1	0	0	{t1,t2,t3}
1	0	0	1	{t1,t2}
0	0	0	0	{t1,t2}
0	1	0	1	{t3}
0	0	1	1	{t3}
1	1	1	1	{t3}

(d)

Fig. 2: Illustrating Algorithm 1 in an example.

the valid t-pairs yet to be covered. The pool initially contains all valid t-pairs (line 2), where λ_t^{τ} refers to the set of valid t-pairs for the test case τ. As t-pairs are covered, they are removed from the pool. At each iteration, we pick the best row $<c, T'>$, such that the configuration c is a valid configuration for all the test cases in $\mathrm{T}' \subseteq \mathrm{T}$, and that there is no other row covering more previously uncovered t-pairs (line 4). The selected row is then included in the test case-aware covering array Ψ_t^M (line 6) and the newly covered t-pairs $\lambda_t^{<c,\mathrm{T}'>}$ in the row are removed from the pool (line 5). When the pool is empty, the iterations terminate (line 3) and the computed test case-aware covering array is returned (line 8).

An integral part of this approach is to choose the best row at each iteration (line 4). In this work, as a proof of concept, we implement this functionality by using Answer Set Programming (ASP).

ASP is a declarative programming paradigm, which represents a computational problem as a "program" whose models, called "answer set", correspond to the solutions [24], [27]. ASP solvers are then used to find the answer sets for the program.

Figure 3 provides an example ASP encoding that picks the best rows in the computation of a 2-way test case-aware covering array for a hypothetical configuration space model. The model contains four binary options `(o1,o2,o3,o4)` and three test case `(t1,t2,t3)`. Below, we explain the encoding in a nutshell with no intention to introduce ASP. For more details about ASP, the interested reader may refer to an introduction [16] or a book [2].

In a configuration, every configuration option must have exactly one valid setting:

```
1 {cfg(Opt,Val) : val(Opt,Val)} 1 :- opt(Opt).
```

A configuration is a valid configuration for a test case `T`, if it is not an invalid configuration for the test case:

```
validCfg(T) :- test(T), not invalidCfg(T).
```

As an example, the encoding provides two test case-specific constraints. Configurations in which `o1=1`, are invalid configurations for the test case `t1`, and the ones in which `o1=0`, are invalid for the test case `t2`:

Algorithm 2 Computing a t-way test case-aware covering array Ψ_t^M by maintaining a single configuration space model M for the software under test, where λ_t^{T} is the set of all t-pairs yet to be covered, λ_t^{τ} is the set of t-pairs yet to be covered for test case τ, and $<c, T'>$ is the best row picked at the current iteration.

```
Input $M$=<$O, V, \mathrm{T}, Q_s, Q_{\mathrm{T}}$>: Config. space model
Input $t$: Covering array strength

1: $\Psi_t^M \leftarrow \emptyset$
2: $\lambda_t^{\mathrm{T}} \leftarrow \bigcup_{\tau \in \mathrm{T}} \lambda_t^{\tau}$
3: while $\lambda_t^{\mathrm{T}} \neq \emptyset$ do
4:     $<c, T'> \leftarrow bestRow(\lambda_t^{\mathrm{T}})$
5:     $\lambda_t^{\mathrm{T}} \leftarrow \lambda_t^{\mathrm{T}} - \lambda_t^{<c,\mathrm{T}'>}$
6:     $\Psi_t^M \leftarrow \Psi_t^M \cup <c, T'>$
7: end while
8: return $\Psi_t^M$
```

```
invalidCfg(t1) :- cfg(o1, 1).
invalidCfg(t2) :- cfg(o1, 0).
```

ASP is expressive enough to encode any type of constraints. The constraints in the example encoding are however kept simple for the sake of the discussion.

The set of valid t-pairs yet to be covered, i.e., λ_t^{T} in Algorithm 2, are expressed as a set of `tpair` facts. For example, the following fact in the encoding indicates that the 2-pair `<<o1=0, o2=1>, t1>` is a valid 2-pair yet to be covered for the test case `t1`:

```
tpair(o1, 0, o2, 1, t1).
```

A 2-pair `<<O1=V1, O2=V2>, T>`, where `O1, O2, V1, V2` and `T` are variables that will be grounded by the ASP solver, is considered to be covered when 1) the 2-tuple `<O1=V1, O2=V2>` appears in a configuration, 2) the configuration is a valid configuration for the test case `T` and `T` is scheduled to be executed in the configuration, and 3) the 2-tuple has not yet been covered for `T`:

```
covered(O1,V1,O2,V2,T) :- cfg(O1,V1), cfg(O2,V2),
                          validCfg(T),
                          tpair(O1,V1,O2,V2,T).
```

Finally, the following ASP directive ensures that the best row that covers the most number of previously uncovered 2-pairs is chosen:

```
#maximize {covered(O1,V1,O2,V2,T)}.
```

Adapting the ASP encoding given in Figure 3 to compute test case-aware covering arrays of arbitrary strength for any given configuration space model is straightforward. To conduct the experiments discussed in Section 5, for example, we developed a simple script, which automatically generated the ASP encoding for a given configuration space model and a value of t.

```
% Test cases
test(t1;t2;t3).

% Config. options and their settings
opt(o1;o2;o3;o4).
val(o1;o2;o3;o4,0;1).

% The definition of a configuration
1 {cfg(Opt,Val) : val(Opt,Val)} 1 :- opt(Opt).

% The definition of a valid config. for a test T
validCfg(T) :- test(T), not invalidCfg(T).

% Example test case-specific constraints:
% t1 skips when o1=1
invalidCfg(t1) :- cfg(o1, 1).
% t2 skips when o1=0
invalidCfg(t2) :- cfg(o1, 0).

% An example 2-pair to cover:
tpair(o1, 0, o2, 1, t1).
%...

% The definition of a covered t-pair
covered(O1,V1,O2,V2,T) :- cfg(O1,V1), cfg(O2,V2),
                          validCfg(T),
                          tpair(O1,V1,O2,V2,T).

% The optimization criterion
#maximize {covered(O1,V1,O2,V2,T)}.
```

Fig. 3: An example ASP encoding for computing the "best" row at a given iteration.

Figure 4 illustrates Algorithm 2 in our running example introduced in Section 4.1. Unlike Algorithm 1, Algorithm 2 maintains a single configuration space model for $t1, t2$, and $t3$. Furthermore, Algorithm 2 keeps a global view of all the 2-pairs to be covered and proceeds by selecting the best row, such that the number of newly covered 2-pairs is maximized at each iteration. The 2-way test case-aware covering array generated by Algorithm 2 as well as the number of newly covered 2-pairs by each row included in the array, can be found in the figure. The first row of the array covers the maximum number of 2-pairs that can be covered by a row (i.e., $\binom{4}{2} * 3 = 18$). As new configurations are included in the array, since the number of remaining 2-pairs to be covered decreases, the number of newly covered 2-pairs monotonically decreases. In the end, the computed 2-way test case-aware covering array requires 6 configurations and 17 test runs.

Comparing the test case-aware covering arrays created by Algorithm 1 and Algorithm 2, we observe that Algorithm 2 increased the number of configurations shared by the test cases. For example, in the test case-aware covering array created by Algorithm 2 (Figure 4), 5 configurations were shared by all of the 3 test cases, whereas in the test case-aware covering array created by Algorithm 1 (Figure 2), only 3 configurations were shared by all the test cases. This, in turn,

Configuration Space Model

```
[options]
   o1,o2,o3,o4: {0, 1}

[constraints]
   t1, t2: #empty
   t3: ¬(o1=0 ∧ o4=0)
         ∧¬(o2=0 ∧ o3=0)
```

Test Case-Aware Covering Array

o1	o2	o3	o4	tests	# of newly covered 2-pairs
1	1	1	1	{t1,t2,t3}	18
0	1	0	1	{t1,t2,t3}	15
1	0	1	0	{t1,t2,t3}	15
0	0	1	1	{t1,t2,t3}	9
1	1	0	0	{t1,t2,t3}	9
0	0	0	0	{t1, t2}	4

Fig. 4: Illustrating Algorithm 2 in an example.

reduced the total number of configurations required from 8 to 6 (i.e., by 25%).

4.3 Algorithm 3: Minimizing number of test runs

Algorithm 1 and 2 aim to minimize the number of configurations included in test case-aware covering arrays. To do that, both algorithms force the test cases to share configurations. However, reducing the number of configurations does not necessarily reduces the number of test runs required. In fact, there is often a trade-off between minimizing the number of configurations and minimizing the number of test runs. The reason behind this trade-off is a simple one. Forcing test cases to share configurations, thus reducing the number of configurations, can cause a test case to execute in a number of configurations to cover a certain set of t-tuples, which could have been otherwise covered by a smaller number of configurations, thus reducing the number of test runs, if the test case was not forced to share the configurations. On the other hand, if configurations are not shared among the test cases, the number of configurations required will naturally increase.

For instance, the brute-force search of the configuration space used in our running example depicted in Figure 2 and 4, proved that a minimum of 6 configurations are needed to satisfy the coverage requirements of all the test cases. The brute-force search also revealed that when the number of configurations is fixed at 6 (i.e., at the minimum), a minimum of 17 test runs are required. From this perspective, the 2-way test case-aware covering array computed by

Algorithm 3 Minimizing the number of test runs required, where Ψ_t^M is the t-way test case-aware covering array to be computed for the configuration space model M, and S_τ and $\Omega_t^{M_\tau}$ are the seed and the traditional t-way covering array created for test case τ, respectively.

Input M=<$O, V, \mathrm{T}, Q_s, Q_\mathrm{T}$>: Config. space model
Input t: Covering array strength

```
1: $\Psi_t^M \leftarrow \emptyset$
2: for each test case $\tau$ in $\mathrm{T}$ do
3:     $S_\tau \leftarrow \emptyset$
4:     $\Omega_t^{M_\tau} \leftarrow \prod(t, M_\tau, S_\tau)$
5:     $\Psi_t^M \leftarrow \Psi_t^M \bullet \Omega_t^{M_\tau}$
6: end for
7: return $\Psi_t^M$
```

Algorithm 2 is optimal. On the other hand, we know that a minimum of 5 configurations are needed for each test case to satisfy the coverage requirement of the test case. This indicates that the minimum number of total test runs needed is 15 ($3*5=15$). However, both Algorithm 1 and Algorithm 2 required more test runs; Algorithm 1 required 16 test runs, whereas Algorithm 2 required 17 test runs.

The algorithm presented in this section, namely Algorithm 3, aims to minimize the number of test runs. To this end, we slightly modify Algorithm 1, such that, instead of creating a seed for each test case, thus forcing the test cases to share configurations, we use an empty seed (line 3). This provides each test case with freedom to select its own configurations (line 4). The rest of the algorithm is the same as Algorithm 1.

At each iteration of the algorithm, since the traditional covering array generator $\prod$ "minimizes" the number of configurations required for a test case, the number of test runs required for each test case would be minimized, and so would the total number of test runs.

Figure 5 illustrates Algorithm 3 in our running example. As is the case with Algorithm 1, a separate configuration space submodel is maintained for each test case, a traditional 2-way covering array is computed for each submodel, and the newly computed covering arrays are merged to construct the 2-way test case-aware covering array. However, unlike Algorithm 1, the seeds are always empty. As Figure 5a-c indicate, the size of the traditional covering array created for each test case is 5 (i.e., at the minimum). Although no attempt is made to share configurations, 3 configurations happen to be shared. In the end, the computed test case-aware covering array (Figure 5d) requires 12 configurations and 15 test runs.

Note that, had the test cases *t1* and *t2* shared exactly the same set of configurations, the number of configurations would have been reduced to 8 without

Configuration Space Submodel for t1:

```
[options]
  o1,o2,o3,o4: {0, 1}

[constraints]
  #empty
```

Seed:
#empty

Covering Array for t1:

o1	o2	o3	o4
1	0	1	0
0	1	1	1
1	1	0	0
1	0	0	1
0	0	0	0

(a)

Configuration Space Submodel for t2:

```
[options]
  o1,o2,o3,o4: {0, 1}

[constraints]
  #empty
```

Seed:
#empty

Covering Array for t2:

o1	o2	o3	o4
0	1	0	1
0	0	1	0
1	0	0	0
1	0	1	1
1	1	1	0

(b)

Configuration Space Submodel for t3:

```
[options]
  o1,o2,o3,o4: {0, 1}

[constraints]
  ¬(o1=0 ∧ o4=0)
    ∧¬(o2=0 ∧ o3=0)
```

Seed:
#empty

Covering Array for t3:

o1	o2	o3	o4	
1	1	0	0	*
1	0	1	0	*
0	1	0	1	*
0	0	1	1	
1	1	1	1	

(c)

Test Case-Aware Covering Array

o1	o2	o3	o4	tests
1	1	0	0	{t1,t3}
1	0	1	0	{t1,t3}
0	1	0	1	{t2,t3}
0	1	1	1	{t1}
1	0	0	1	{t1}
0	0	0	0	{t1}
0	0	1	0	{t2}
1	0	0	0	{t2}
1	0	1	1	{t2}
1	1	1	0	{t2}
0	0	1	1	{t3}
1	1	1	1	{t3}

(d)

Fig. 5: Illustrating Algorithm 3 in an example.

changing the number of test runs required. However, we opted not to share any configurations between *t1* and *t2* to demonstrate potential scenarios that could arise when non-deterministic traditional covering array generators are used.

In any case, brute-force search proved that when the number of test runs is fixed at 15 (i.e., at the minimum), a minimum of 7 configurations are required. That is, for our running example, minimizing the number of configurations requires a minimum of 6 configurations and 17 test runs, whereas minimizing the number of test runs requires a minimum of 7 configurations and 15 test runs, demonstrating the potential trade-off between the two optimization criteria.

5 Experiments

We conducted a series of empirical studies to 1) gain more insight on test case-specific constraints occurring in practice, 2) quantify the extent to which traditional covering arrays suffer from ignoring such constraints, and 3) evaluate the efficiency and effectiveness of the test case-aware covering arrays.

In these studies, we used two widely-used highly-configurable software systems as our subject applications: Apache v2.3.11-beta and MySQL v5.1. Apache is a HTTP server. MySQL is a database management system.

We chose these systems for several reasons. First, they share the key characteristics common to configurable software systems. They are highly configurable with dozens of configuration options supporting a wide variety of features. They have a large code base and substantial test code. Both systems enjoy a large developer community that actively updates and tests the systems. Second, like many configurable software systems, developers of these systems cannot exhaustively test the entire configuration space; the number of possible configurations is far beyond the resources available to run the test cases in a timely manner, e.g., for regression testing.

All the experiments were performed on an AMD 64 Athlon machine with 4 GB of RAM, running Ubuntu 10.10 operating system.

5.1 Study 1: Test case-specific constraints in practice

In the first study, our goal was to gain more insight on test case-specific constraints occurring in practice.

5.1.1 Study setup

To conduct the study, we studied the test cases that came with the source code distribution of our subject applications. Each test case in both subject applications had its own test oracle, which determines whether each test case execution “passed”, “failed”, or was “skipped”. Successful test cases simply emit pass. Failed test cases emit fail. A test case returns skipped when it determines that it cannot run in a given configuration, indicating that the test case has a test case-specific constraint.

TABLE 1: Traditional configuration space model used in the experiments for Apache.

option	settings
case-filter	{enable, disable}
ssl	{enable, disable}
dav	{enable, disable}
echo	{enable, disable}
rewrite	{enable, disable}
case-filter-in	{enable, disable}
bucketeer	{enable, disable}
info	{enable, disable}
headers	{enable, disable}
vhost-alias	{enable, disable}
cgi	{enable, disable}
proxy-http	{enable, disable}
proxy	{enable, disable}
system-wide constraint	
proxy-http = enable $\rightarrow$ proxy=enable	

TABLE 2: Traditional configuration space model used in the experiments for MySQL.

option	settings
log-format	{row, statement, mixed}
sql-mode	{strict, traditional, ansi}
ext-charsets	{disable, complex, all}
innodb	{enable, disable}
libedit	{enable, disable}
log-bin	{enable, disable}
readline	{enable, disable}
ndbcluster	{enable, disable}
ssl	{enable, disable}
archive	{enable, disable}
blockhole	{enable, disable}
federated	{enable, disable}
system-wide constraint	
ssl=disable[1] $\wedge$ (libedit=enable $\rightarrow$ readline=disable)	

5.1.2 Data and analysis

We first observed that the test case-specific constraints were encoded in the test oracles of our subject applications, indicating that they were known to the developers. This is important, because the more accurate and complete the test case-specific constraints are, the better the test case-aware covering arrays perform in avoiding masking effects.

We then determined that, out of 3789 and 738 test cases examined for Apache and MySQL, respectively, 378 Apache test cases and 337 MySQL test cases had some test case-specific constraints. This is a good indicator that these systems (and potentially other configurable systems) can benefit from test case-aware covering arrays in practice.

To identify the actual test case-specific constraints for these configuration-dependent test cases, we studied the test cases and their oracles, read the user manuals, and conducted experiments as needed. It turned out that the test case-specific constraints together with the system-wide constraints needed to build the systems, involved a total of 13 and 12 unique configuration options for Apache and MySQL, respectively. These configuration options and their settings together with the system-wide constraints can be found in Table 1 and 2.

We observed that the configuration-dependent test cases formed clusters with respect to their test case-specific constraints. In other words, we had clusters of test cases having exactly the same constraints. We identified 17 clusters for Apache and 30 clusters for MySQL. Table 3 and 4 present the the distribution of the test cases over the clusters, each of which is identified by a unique test case-specific constraint. The constraints in these tables express the conditions that must be satisfied for the test cases to run. For example, the first row in Table 4 depicts that 86 MySQL test cases share exactly the same constraint

`log-bin=enable` $\wedge$ `sql-mode`$\neq$ `ansi`,

indicating that these test cases skip all configurations in which `log-bin`$\neq$ `enable` or `sql-mode=ansi`.

An in-depth analysis revealed that these clusters were formed due to two factors. First, there were some test cases specifically designed to test features that need to be explicitly configured into the system. For example, all of the 16 test cases in cluster 6 (Table 3) were designed to test the Distributed Authoring and Versioning (DAV) feature of the Apache server. Therefore, these test cases simply assume that DAV is already configured into the system (i.e., `dav=enable`). If not, they refuse to run.

The second factor was that some test cases originally designed to test a wide range of features, were dependent on the same configurable feature. For example, a number of test cases in MySQL used the SQL `'LIKE'` operator in their database queries with wildcard characters that are not supported by the ANSI SQL standard. Consequently, these test cases refuse to run when the system is configured with the ANSI SQL query engine (i.e., `sql-mode=ansi`).

5.1.3 Discussion

Considering the nature of the factors leading to having clusters of test cases sharing exactly the same constraints, it is likely that other configurable systems exhibit the same tendency. This is a valuable observation towards improving the scalability of test case-aware covering array generators in practice. One test case can be chosen from each cluster and the test case-aware covering array can be created only for the selected test cases rather than for all test cases. Once the test case-aware covering array is computed, each test case in the array can then be replaced with all the test cases in the respective cluster. As the number of t-pairs is proportional to the number of test cases, the proposed approach can considerably reduce the number of t-pairs that need to be dealt with

1. This constraint was added to avoid some failures we consistently experienced during the experiments. Although the failures appeared to be platform-specific, to carry out the experiments, we opted to express the constraint as a system-wide constraint.

TABLE 3: Test case-specific constraints used in the experiments for Apache.

cluster idx	# of tests	test case-specific constraint
1	172	ssl=enable ∧ proxy-http=enable
2	74	ssl=enable
3	26	rewrite=enable
4	22	headers=enable
5	21	proxy=enable
6	16	dav=enable
7	11	case-filter=enable
8	8	vhost-alias=enable
9	7	proxy-http=enable
10	5	proxy=enable∧rewrite=enable ∧ cgi=enable
11	4	echo=enable
12	3	ssl=enable ∧ headers=enable
13	2	rewrite=enable ∧ proxy=enable
14	2	ssl=enable ∧ case-filter-in=enable
15	2	case-filter-in=enable
16	2	bucketeer=enable
17	1	info=enable

at runtime, thus improving scalability. We employed this approach to compute the test case-aware covering arrays studied in Section 5.3.

5.2 Study 2: Masking effects in traditional covering arrays

In the second study, our goal was to quantify the extent to which traditional covering arrays suffer from ignoring test case-specific constraints.

5.2.1 Study setup

To conduct the study, we mimicked the way traditional covering arrays are used in practice. In particular, given a configuration space model, we created traditional covering arrays, scheduled all the test cases to execute in all the selected configurations, and quantified the consequences of masking effects caused by the overlooked test case-specific constraints.

We first started with the configuration space models given in Table 1 and 2. These models consist of only those configuration options that are referenced by the system-wide or test case-specific constraints. In the remainder of the paper, a configuration option that is referenced by a constraint is referred to as a *constrained option*, e.g., all the options in our initial configuration models were constrained options.

We then varied the percentage of constrained options in configuration space models (in short, *cop*) to evaluate the consequence of *cop* on masking effects. Given a value of *cop*, our initial configuration models were augmented with unconstrained binary configuration options to obtain the desired percentage level. For example, to obtain 60% constrained options (i.e., cop=60) for Apache, 9 unconstrained binary options were added to the initial configuration space model

TABLE 4: Test case-specific constraints used in the experiments for MySQL.

cluster idx	# of tests	test case-specific constraint
1	86	log-bin=enable ∧ sql-mode≠ansi
2	60	ndbcluster=enable
3	33	innodb=enable
4	28	log-format≠row ∧ log-bin=enable ∧ sql-mode≠ansi
5	22	sql-mode≠ansi
6	18	ext-charsets≠disable ∧ sql-mode≠ansi
7	17	log-format≠statement ∧ log-bin=enable ∧ ndbcluster=enable
8	17	innodb=enable ∧ log-bin=enable ∧ sql-mode≠ansi
9	16	log-bin=enable ∧ ndbcluster=enable
10	6	log-format≠row ∧ innodb=enable ∧ log-bin=enable ∧ sql-mode≠ansi
11	4	log-format≠row ∧ ext-charsets≠disable ∧ log-bin=enable ∧ sql-mode≠ansi
12	4	federated=enable ∧ log-bin=enable ∧ sql-mode≠ansi
13	4	innodb=enable ∧ sql-mode≠ansi
14	4	ndbcluster=enable ∧ sql-mode≠ansi
15	2	log-format≠statement ∧ innodb=enable ∧ log-bin=enable ∧ sql-mode≠ansi
16	2	blackhole=enable ∧ log-bin=enable ∧ ndbcluster=enable
17	1	archive=enable ∧ log-format≠row ∧ log-bin=enable ∧ sql-mode≠ansi
18	1	federated=enable ∧ innodb=enable ∧ log-bin=enable ∧ sql-mode≠ansi
19	1	log-format≠row ∧ blackhole=enable ∧ log-bin=enable ∧ sql-mode≠ansi
20	1	log-format≠statement ∧ log-bin=enable ∧ ndbcluster=enable ∧ sql-mode≠ansi
21	1	ext-charsets≠disable ∧ log-bin=enable ∧ sql-mode≠ansi
22	1	log-bin=enable ∧ ndbcluster=enable ∧ sql-mode≠ansi
23	1	log-format≠row ∧ log-bin=enable ∧ ndbcluster=enable
24	1	ext-charsets≠disable ∧ innodb=enable ∧ sql-mode≠ansi
25	1	innodb=enable ∧ log-bin=enable ∧ ndbcluster=enable
26	1	innodb=enable ∧ ndbcluster=enable
27	1	archive=enable ∧ innodb=enable
28	1	archive=enable
29	1	log-bin=enable
30	1	ext-charsets≠all

given in Table 1. In particular, we experimented with cop=20, 30, 40, 50, 60, 80, and 100.

Given a configuration space model, we created traditional t-way covering arrays with varying strengths ($2 \leq t \leq 5$). This was done by using a well-known covering array generator, called ACTS (v1.r9.3.2) [1]. Once a t-way covering array was generated, all of the 378 test cases for Apache and 337 test cases for MySQL, which are known to have the test case-specific constraints given in Table 3 and 4, were scheduled to be executed in all the selected configurations. Test cases with no test case-specific constraints were

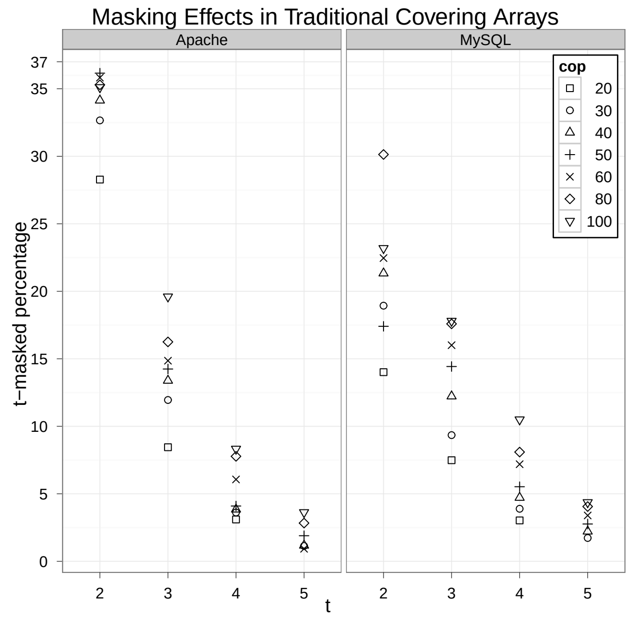


Fig. 6: Masking effects in the traditional covering arrays created for Study 2.

not utilized in the experiments, as they would not suffer from any masking effects.

To quantify the harmful consequences of masking effects, we use a metric, called *t-masked* [15]. This metric counts the number of unique t-pairs that are for sure not tested due to the overlooked test case-specific constraints. If a test case skips a configuration, none of the t-tuples in the configuration will be tested by that test case. To compute the value of t-masked, for each test case, we first count the number of *valid* t-tuples that appear *only* in the configurations skipped by the test case. We then add these counts up for all the test cases. For a given test case, the set of valid t-tuples as well as the configurations skipped, were determined by using the respective system-wide constraints in Table 1 and 2 and the respective test case-specific constraints in Table 3 and 4.

We, furthermore, compute the percentage of valid t-pairs that are masked and refer to it as *t-masked percentage*. For instance, the value of 3-masked and 3-masked percentage in the traditional 3-way covering array given in Figure 1, is 8 and 11, respectively. For adequate testing, t-masked, thus t-masked percentage, should be 0.

5.2.2 Data and analysis

Table 5 and 6 in Appendix A present the results we obtained. The columns in these tables depict the strength of the covering array generated (i.e., t), the number of options in the configuration space model used, the percentage of constrained options in the model (i.e., cop), the number of configurations included in the computed covering array, the number of test runs required by the array, the time it took to construct the array (in minutes), and the values of t-masked and t-masked percentage for various values of t. For each row of the tables, we created between 2 and 10 traditional covering arrays, depending on the computational resources required. The values of independent variables reported in these tables, are the average values obtained from these arrays. Figure 6 visualizes the results we obtained.

We first observed that all the traditional t-way covering arrays created for the study suffered from masking effects caused by the overlooked test case-specific constraints (Figure 6 and Table 5-6). For example, when t=2 and cop=100, about 35% of all the valid 2-pairs, on average, were not tested at all by the traditonal 2-way covering arrays created for Apache, i.e., about $35,250$ 2-pairs were masked, on average.

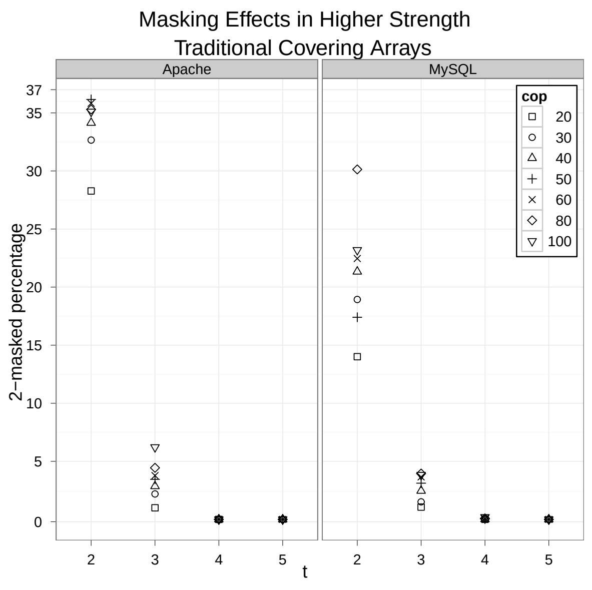
We then observed that, for a fixed configuration space model, higher strength covering arrays suffered relatively less compared to lower strength covering arrays. As t increased, t-masked percentage decreased, but never reached 0, when $2 \leq t \leq 5$ (Figure 6). For instance, when cop=100 for Apache, 5-masked percentage was about 3.6 in 5-way covering arrays as opposed to the 2-masked percentage of about 35 in 2-way covering arrays. Clearly, when t=n, where n is the number of options in the configuration space model, as the configuration space will be tested exhaustively, t-masked percentage will be 0. However, since $t \lll n$ in practice, with t=2 being the most common case, handling test case-specific constraints gains in importance.

We furthermore observed that, for a fixed value of t, as the constrained options percentage (i.e., cop) decreased, t-masked percentage tended to decrease (Figure 6). The trend is more clearly observable when $t > 2$. Clearly, when cop=0, as there will be no test case-specific constraints, t-masked percentage will be 0. However, t-masked percentage never reached 0 for the values of cop used in the experiments.

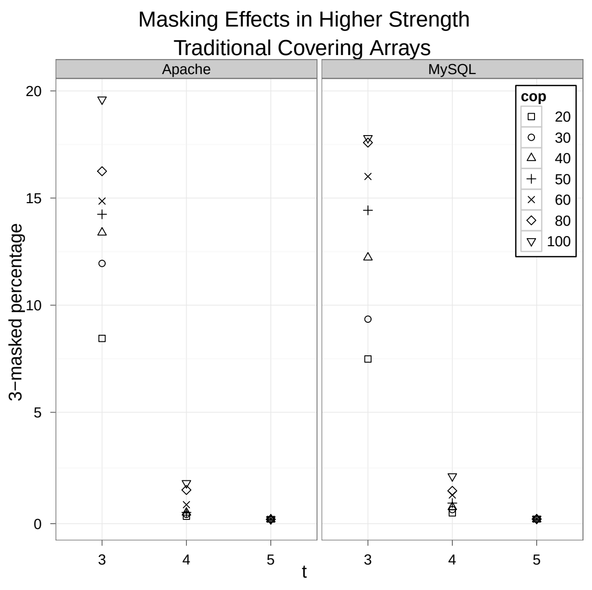
One way to remove masking effects can be to create a higher strength covering array in order to cover lower order interactions among configuration options, i.e., using a t-way covering array to obtain t'-way coverage, where $t' < t$ [15]. For example, using a 3-way covering array is likely to reduce the number of 2-pairs masked compared to using a 2-way covering array, as every 2-tuple will appear multiple times in different 3-tuples.

To evaluate this conjecture, we used the traditional t-way covering arrays created in this study, but rather than monitoring t-masked percentages in these arrays, we monitored 2-masked (when $t \geq 2$) and 3-masked percentages (when $t \geq 3$). Figure 7 visualizes the results we obtained. The raw data can be found in Table 5 and 6.

We observed that, for a fixed configuration space model, as t increased, both 2-masked and 3-masked



(a)



(b)

Fig. 7: Using t-way traditional covering arrays to reduce a) 2-masked (when $t \geq 2$) and b) 3-masked percentages (when $t \geq 3$).

percentages decreased, and rarely reached 0. For example, when cop=100 for Apache, 2-masked percentages were 35.11, 6.22, 0.002, and finally 0 for t=2, 3, 4, and 5, respectively.

Clearly, using a sufficiently high strength traditional t-way covering array can prevent all t'-pairs from being masked ($t' < t$). However, the traditional t-way covering arrays used in the experiments managed to do so at the cost of significantly increased number of configurations and test runs compared to the t'-way test case-aware covering arrays. For instance, when cop=100 for Apache, the traditional 5-way covering arrays, while preventing all 2-pairs from being masked, required about 6 times (684%) and 18 times (1888%) more configurations and test runs, respectively, compared to the 2-way test case-aware covering arrays created by Algorithm 2, which also prevented all 2-pairs from being masked (Table 7 in Appendix B). Further discussion is provided in Section 5.3.

5.2.3 Discussion

In the experiments, we used only those test cases that had some test case-specific constraints. In the remainder of the paper, such test cases are referred to as *constrained test cases*, whereas test cases with no test case-specific constraints are referred to as *unconstrained test cases*.

Had we used some unconstrained test cases in our experiments, regardless of the number of such test cases used, t-masked values observed in the experiments would not have changed, i.e., the same number of t-pairs would have been masked. This is because test cases are independent of each other and unconstrained test cases do not suffer from any masking effects. On the other hand, t-masked percentages observed in the experiments would have decreased as the percentage of constrained test cases in the test suite decreased. This is because adding unconstrained test cases increases the total number of t-pairs to be covered without affecting the number of t-pairs masked; the dividend stays the same, but the divisor increases.

As an example, consider the traditional 2-way covering arrays created for MySQL with cop=100. When we used 337 constrained MySQL test cases in the experiments (i.e., when 100% of the test cases were constrained), out of $88,328$ valid 2-pairs about $20,483$ 2-pairs were masked; 2-masked percentage was about 23. On the other hand, Figure 8 plots 2-masked percentages that were obtained by varying the percentage of constrained test cases in the test suite. The variations in the test suite were obtained by keeping the 337 constrained test cases in the suite and adding new unconstrained test cases as needed. Regardless of the number of unconstrained test cases added, the 2-masked value was always $20,483$. However, adding a new unconstrained test case increased the total number of valid 2-pairs by 307. Thus, 2-masked percentage decreased accordingly.

Note that, since t-masked values do not depend on the number of unconstrained test cases, all the comparative analyses conducted in Section 5.2.2 are valid regardless of the presence or absence of unconstrained test cases. However, the real values of t-masked percentages may vary. In the analyses, we opted to use t-masked percentages rather than t-masked values, since, being a normalized metric, t-

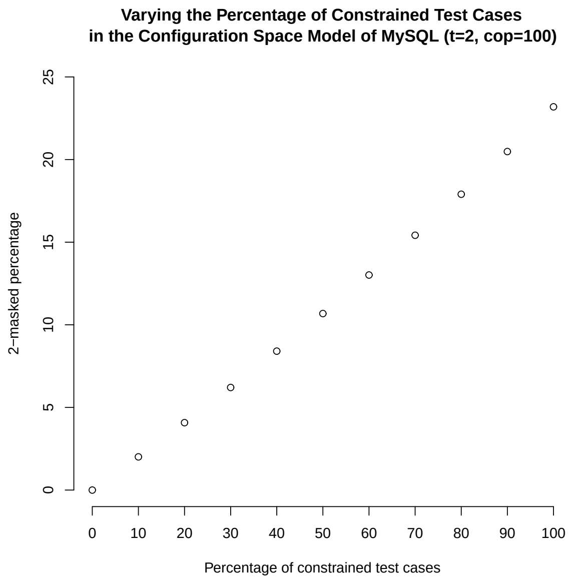


Fig. 8: Varying the percentage of constrained test cases in the configuration space model of MySQL (t=2 and cop=100).

masked percentage offers a better interpretation of the results. For example, for a fixed value of cop, when t increased ($2 \leq t \leq 5$), t-masked values increased, but, as the total number of valid t-pairs increased at a faster rate, t-masked percentages decreased, indicating that higher strength covering arrays actually suffered less.

Overall, our analysis results suggest that the extent to which traditional covering arrays suffer from masking effects depends on many factors, such as the strength of the covering array, the percentage of constrained options in the configuration space model, and the percentage of constrained test cases in the test suite. Therefore, given a particular combinatorial testing scenario, reliably estimating the consequences of masking effects caused by the overlooked test case-specific constraints is of practical value. To do that without ever configuring the system under test and without ever running any test cases, we give the following guidelines to the users of covering arrays: Generate a traditional t-way covering array for the scenario at hand and then compute the t-masked and t-masked percentage values. The larger the value of t-masked and/or the value of t-masked percentage, the more the covering array suffers. Both t-masked and t-masked percentage values are important for the evaluation. For example, when t=5 and cop=30 for MySQL, 5-masked percentage was 1.7. Although this may seem to be a small percentage, it indicates that quite a large number of 5-pairs, i.e., more than 120 million 5-pairs, were not tested at all. The evaluation results can then be used to decide if test case-aware covering arrays are required for the scenario at hand.

5.3 Study 3: Test case-aware covering arrays

In the third study, our goal was to evaluate the effectiveness and efficiency of test case-aware covering arrays in avoiding masking effects.

5.3.1 Study setup

To carry out the study, we first implemented the algorithms given in Section 4. In the implementation of Algorithm 1 and 3, we used *ACTS v1.r9.3.2* [1] as our traditional covering array generator. To implement Algorithm 2, we used *gringo v3.0.3*, a grounder for ASP programs, together with *clasp v2.0.2*, an ASP solver. The solver was configured to have a 3-minute timeout period, i.e., at each iteration of the algorithm, the best row found in 3 minutes was returned.

We then populated the traditional configuration space models used in Study 2 with the test case-specific constraints identified in Study 1. For a given configuration space model and a value of t, we created between 2 and 10 t-way test case-aware covering arrays ($2 \leq t \leq 3$), depending on the computational resources required. The values reported are the average values obtained from these arrays.

Furthermore, to evaluate the cost of using test case-aware covering arrays, we in this section use a cost function. In practice, typically, once a covering array is constructed, it is repeatedly used for testing as long as the underlying configuration space model stays the same. Therefore, we define our cost function as follows:

$$cost = c_{ca} + M(N_{ca}c_c + R_{ca}c_r), \qquad (1)$$

where c_{ca}, c_c, and c_r are the cost of computing the covering array ca (either a test case-aware or a traditional covering array), the average cost of configuring the system under test with a given configuration, and the average cost of running a single test case, respectively. Furthermore, N_{ca} and R_{ca} are the number of configurations and the number of test runs required by the covering array ca, whereas M is the number of times the array is used for testing.

For a given configuration space model, when the goal is to minimize the number of configurations (i.e., when $c_r \approx 0$, which is implicitly assumed by this optimization criterion), a test case-aware covering array tca is more cost-effective than a traditional covering array ca when

$$c_{tca} + MN_{tca}c_c < c_{ca} + MN_{ca}c_c. \qquad (2)$$

Assuming that $c_{tca} > c_{ca}$, but $N_{tca} < N_{ca}$, we obtain

$$\frac{c_{tca} - c_{ca}}{c_c(N_{ca} - N_{tca})} < M. \qquad (3)$$

Similarly, when the goal is to minimize the number of test runs (i.e., when $c_c \approx 0$, which is implicitly

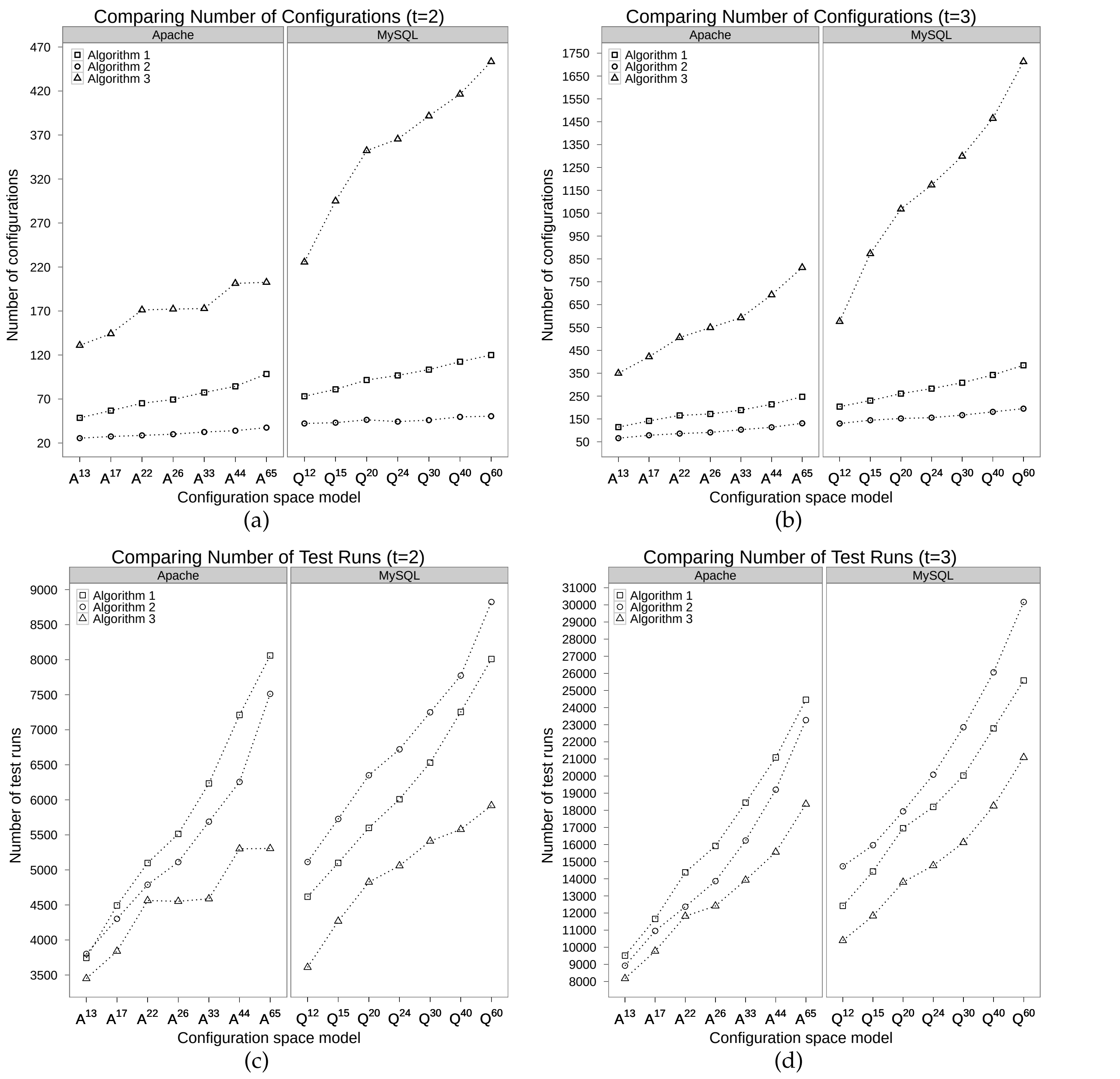


Fig. 9: Comparing the number of configurations (a-b) and test runs (c-d) required by the test case-aware covering arrays.

assumed by this optimization criterion), tca is more cost-effective than ca when

$$\frac{c_{tca} - c_{ca}}{c_r(R_{ca} - R_{tca})} < M, \qquad (4)$$

assuming that $c_{tca} > c_{ca}$, but $R_{tca} < R_{ca}$.

We refer to the minimum M value satisfying inequality (3) or inequality (4) as $\hat{M}$. $\hat{M}$ is the break-even point – that is, the number of times testing is to be performed, where the overall costs of of testing with the test case-aware covering array tca are equal to the overall costs of testing with the traditional covering array ca. If testing is to be performed on more occasions, testing with the test case-aware covering array will be more cost-effective. The lower the value of $\hat{M}$, the broader the range of circumstances in which the test case-aware covering array is more cost effective; if it is less than 1, the test case-aware covering array is always the more cost-effective choice.

In the experiments, we computed the c_c and c_r costs for our subject applications by building the subject applications with their default configurations and then executing the respective test suites. We opted to use the default configurations for several reasons. First, since both subject applications have a large configuration space, it was infeasible for us to test the entire configuration space to compute the actual average costs. Second, we observed that a large portion of the

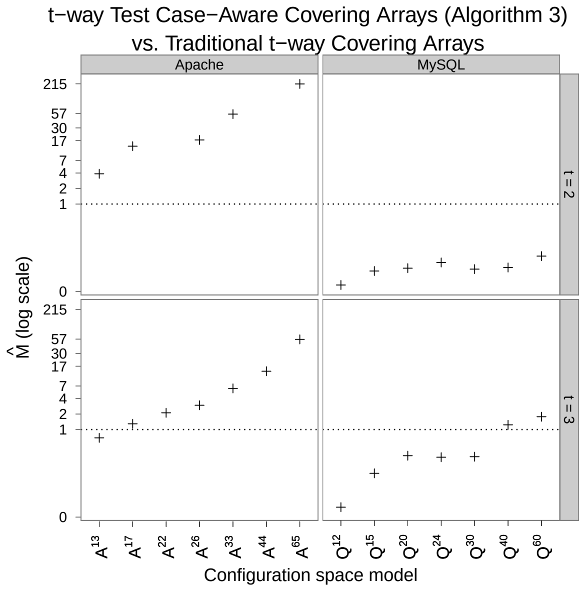


Fig. 10: Comparing the cost of using t-way test case-aware covering arrays created by Algorithm 3 to that of traditional t-way covering arrays.

features that are costly to build and to test, were not included in the default configurations. This prevented us, to the extent possible, from overestimating the costs. It turned out that, for Apache, c_c=8 minutes for building the entire software suite and c_r=0.00058 minutes, and, for MySQL, c_c=82 minutes for building the entire software suite and c_r=0.055 minutes.

5.3.2 *Data and analysis*

We first compared the test case-aware covering arrays created by our algorithms, with each other. Figure 9 visualizes the results we obtained. In these figures, the horizontal axis denotes configuration space models. The literals `A` and `Q` refer to the configuration space models of Apache and MySQL, respectively. The superscript numbers depict the number of configuration options used in the models. The vertical axis denotes the number of configurations in Figure 9a-b and the number of test runs in Figure 9c-d. The raw data can be found in Table 7 and 8 in Appendix B.

Algorithm 1 and Algorithm 2 aim to minimize the number of configurations. Comparing these algorithms (Figure 9a-b), we observed that having a global view of t-pairs (i.e., Algorithm 2) performed better than having a partial view (i.e., Algorithm 1). Algorithm 2, compared to Algorithm 1, reduced the number of configurations by 54% when t=2 and by 45% when t=3, on average.

Algorithm 2, however, provided these improvements at the cost of increased construction time. For instance, when cop=20 for Apache, our implementation of Algorithm 1 took 291 minutes on average to create a 3-way test case-aware covering array, whereas that of Algorithm 2 took 1476 minutes, but reduced the number of configurations by 47%. Therefore, the total cost depends on the actual cost of testing, e.g., 47% reductions in the number of configurations may or may not compensate for the increase in the construction time.

We conjecture however that, in regression testing scenarios, where the same covering array is repeatedly used for testing, the increase in the construction cost is likely to be compensated by the decrease in the actual cost of testing. Therefore, Algorithm 2 is of interest, especially in regression testing scenarios. For example, for each subject application using inequality (3) revealed that, in 13 out of 14 comparisons (7 values of cop $\times$ 2 values of t) made between Algorithm 1 and Algorithm 2, Algorithm 2 was more cost-effective than Algorithm 1 after only a single use of the computed test case-aware covering arrays, i.e., $\hat{M} < 1$. In the remaining comparison, $1 < \hat{M} < 2$ for both subject applications, requiring to use the computed array at least twice for Algorithm 2 to become more cost-effective.

Algorithm 3, on the other hand, aim to minimize the number of test runs. Comparing Algorithm 3 with Algorithm 1 and 2, we observed that Algorithm 3 reduced the number of test runs by 20% when t=2 and by 21% when t=3 (Figure 9c-d), on average, while increasing the number of configurations by 383% and by 400%, respectively (Figure 9a-b). Although there is a profound difference between the percentage reduction in the number of test runs and the percentage reduction in the number of configurations, reducing the number of test runs, thus Algorithm 3, is still of great importance when the cost of configuring the system is negligible.

We then compared t-way test case-aware covering arrays with traditional t-way covering arrays. When the goal was to minimize the number of test runs, we observed that the t-way test case-aware covering arrays created by Algorithm 3 (i.e., our best performing algorithm for reducing the number of test runs), removed all masking effects by using fewer or comparable number of test runs compared to the traditional t-way covering arrays. The t-way test case-aware covering arrays computed by Algorithm 3, except for 2 cases, which required 0.6% and 0.2% increase in the test runs, reduced the number of test runs by 11% when t=2 and by 17% when t=3, on average, compared to the traditional t-way covering arrays.

These test case-aware covering arrays, however, took more time to construct than the traditional covering arrays. Comparing the total costs by using inequality (4), we observed that, for MySQL, $\lceil \hat{M} \rceil$ was 1 in 12 out of 14 comparisons and 2 in the remaining 2 comparisons (Figure 10). For Apache, except for the 2 cases where the test case-aware covering arrays required slightly more test runs, $\lceil \hat{M} \rceil$ was less than 18 in 9 out of 12 comparisons (min: 1, avg: 33, and

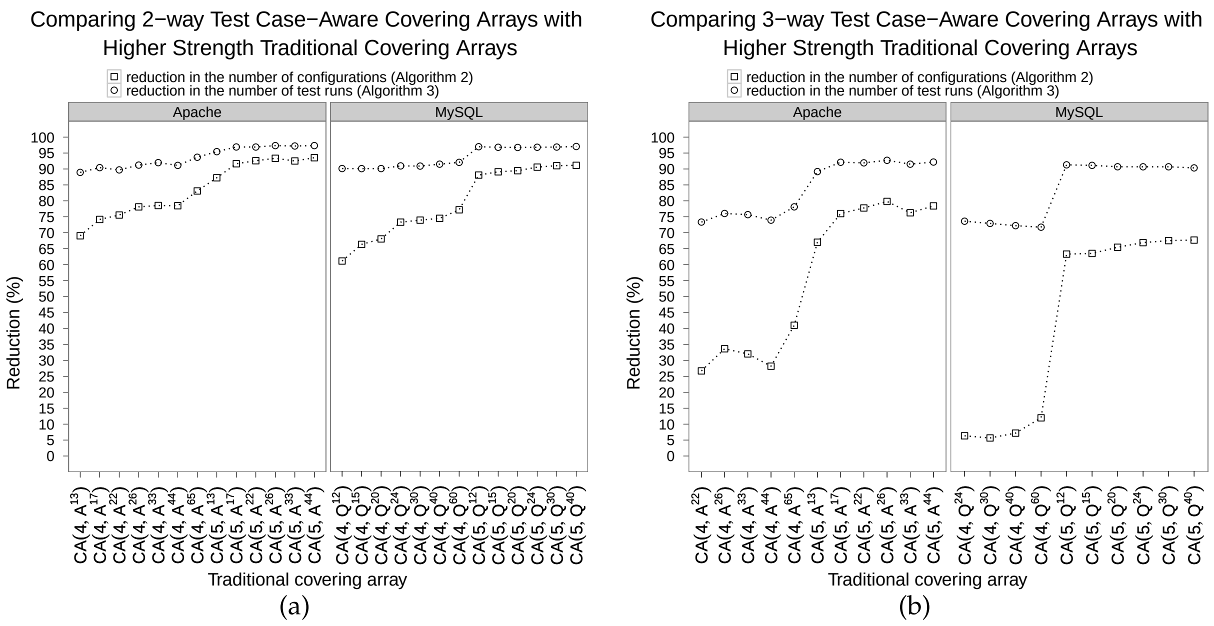


Fig. 11: Comparing a) 2-way b) 3-way test case-aware covering arrays with higher strength traditional covering arrays.

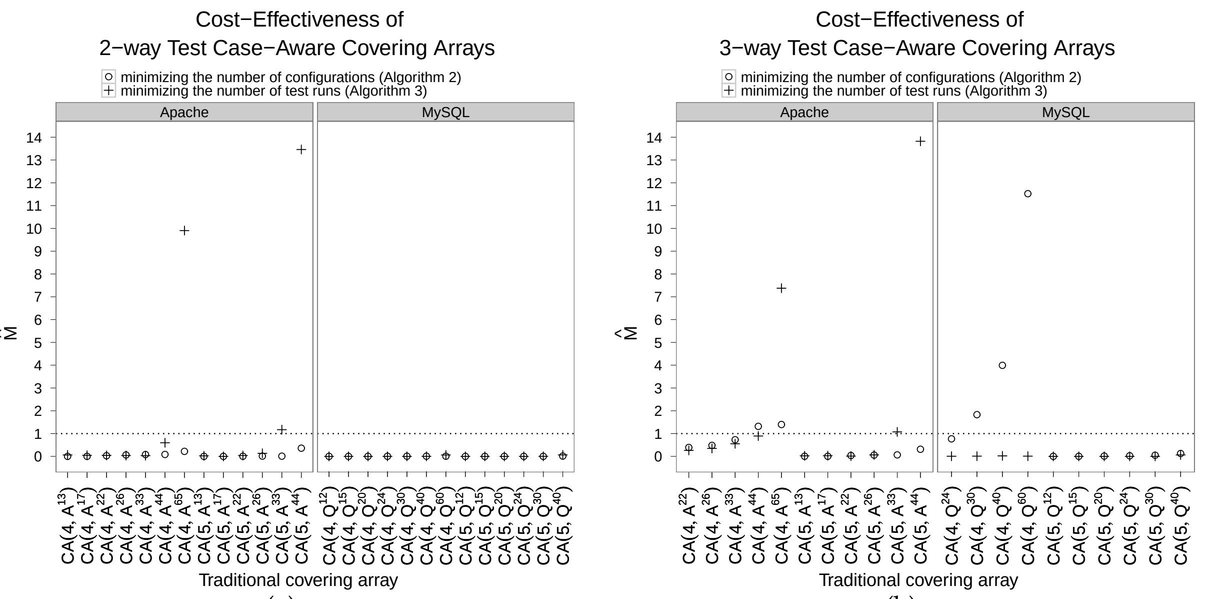
max: 215). One reason we observed high $\hat{M}$ values for Apache was the small c_r cost (i.e., c_r=35 milliseconds) on this subject application, which is not aligned well with the cost model Algorithm 3 is designed for. Had we had c_r=1 second, for example, $\lceil \hat{M} \rceil$ would have been 1 in 9 comparisons, 2 in 2 comparisons, and 8 in 1 comparison.

On the other hand, when the goal was to minimize the number of configurations, the t-way test case-aware covering arrays required more configurations than the traditional t-way covering arrays. For example, the t-way test case-aware covering arrays created by Algorithm 2 required 158% more configurations when t=2 and 174% more configurations when t=3, on average, than the traditional t-way covering arrays. As discussed in Section 3, removing masking effects is likely to increase the number of configurations required, as the t-pairs being masked in traditional covering arrays may need to be covered in additional configurations. Therefore, to evaluate the efficiency provided by test case-aware covering arrays, we compared them to traditional higher strength covering arrays – the best candidate that we know of to compare our results. In particular, we compared the t'-way test case-aware covering arrays with the traditional t-way covering arrays that prevented all or almost all t'-pairs from being masked, where $t' < t$.

To carry out the analysis, among all the traditional covering arrays created in Study 2, we first picked the ones with a 2-masked (t'=2) or 3-masked (t'=3) percentage of less than 1 and determined the configuration space models used to create these traditional covering arrays. We then used our algorithms to create 2-way and 3-way test case-aware covering arrays for the same configuration space models, depending on the value of t'. Finally, we compared the sizes of the traditional covering arrays to those of the test case-aware covering arrays.

Figure 11 presents the percentage reductions obtained by the 2-way and 3-way test case-aware covering arrays. In this figure, the horizontal axis depicts the traditional covering arrays used in the comparisons. The first and the second item in the parentheses indicate the value of t and the configuration space model used, respectively. The vertical axis denotes the percentage reductions obtained. The symbol '□' represents the percentage reductions in the number of configurations provided by Algorithm 2 (i.e., our best performing algorithm for reducing the number of configurations), whereas the symbol '○' represents the percentage reductions in the number of test runs provided by Algorithm 3 (i.e., our best performing algorithm for reducing the number of test runs).

For instance, when t'=2, the first tick on the horizontal axis in Figure 11a indicates that the traditional 4-way covering arrays (t=4) created for the configuration space model containing 13 Apache options, performed well in preventing 2-pairs from being masked; less than 1 percent of all valid 2-pairs (more accurately, 0.002% of the 2-pairs) were masked, on average (Table 5). These traditional 4-way covering arrays required 82.40 configurations and 31,147.20 test runs, on average. On the other hand, when the goal was to minimize the number of configurations, our 2-way test case-aware covering arrays created by Algorithm 2 for the same set of configuration options, required 25.50 configurations on average (Table 7). Similarly, when the goal was to minimize the number



(a) (b)

Fig. 12: Comparing the cost of using a) 2-way b) 3-way test case-aware covering arrays to that of higher strength traditional covering arrays.

of test runs, our 2-way test case-aware covering arrays created by Algorithm 3, required $3,449.60$ test runs on average (Table 7). That is, compared to the traditional 4-way covering arrays, which prevented almost all (but not all) 2-pairs from being masked, Algorithm 2 reduced the number of configurations by 69% and Algorithm 3 reduced the number of test runs by 89%, while preventing all 2-pairs from being masked.

Overall, compared to the higher strength traditional covering arrays used in Figure 11, the 2-way and 3-way test case-aware covering arrays created by Algorithm 2, while not suffering from any masking effects, reduced the number of configurations by 82% and by 50%, on average, respectively. Similarly, the 2-way and 3-way test case-aware covering arrays created by Algorithm 3, while not suffering from any masking effects, reduced the number of test runs by 94% and by 84%, on average, respectively.

However, in almost all the comparisons made in Figure 11, constructing the test case-aware covering arrays required more time than constructing the respective higher strength traditional covering arrays. Comparing the total costs by using inequality (3) and (4), we observed that, when the goal was to minimize the number of test runs, in 87% (41 out of 47) of the comparisons, Algorithm 3 was more cost-effective than the traditional higher strength covering arrays after only a single use of the arrays, i.e., $\hat{M} < 1$ (Figure 12). For the rest of the comparisons, the maximum value of $\hat{M}$ was 13.46. When the goal was to minimize the number of configurations, in 89% (42 out of 47) of the comparisons, Algorithm 2 required only a single use of the generated test case-aware covering arrays to become more cost-effective than the traditional covering arrays. For the rest of the comparisons, the maximum value of $\hat{M}$ was 11.52.

To interpret the $\hat{M}$ values obtained in this study, consider a daily build scenario where the same covering array is used once a day for smoke testing the configuration spaces of our subject applications. Had our test case-aware covering arrays been used in such a scenario, they, in the worst case, would have compensated for their construction cost in 14 days, compared to the higher strength covering arrays. We conjecture that, as the configuration space models of our subject applications are highly unlikely to change in every 14 days (they have stable code bases), the test case-aware covering arrays would have been more cost-effective than their counterparts. Furthermore, $\hat{M}$ is inversely proportional to c_c and c_r (inequality 3 and 4). Therefore, the more the cost of configuring the system and/or the more the cost of running the test cases, the faster test case-aware covering arrays compensate for their construction cost.

5.3.3 *Discussion*

In any case, reducing the construction cost of test case-aware covering arrays is still of practical concern. As discussed in Section 7, there are 4 main categories of methods to compute traditional covering arrays; random search-based methods [28], heuristic search-based methods [7], [10], [12], [19], [29], mathematical methods [20], [21], [32], [33], and greedy methods [4], [6], [8], [9], [13], [23], [30], [31]. We conjecture that all of these methods can also be used to compute test case-aware covering arrays. However, we opted to leverage only the greedy algorithmic paradigm due to the relative ease of its application. Furthermore, all the algorithms presented in this work as well as their implementations are developed mainly for correctness, not for runtime performance. As a future work, we plan to study alternative construction methods, such as heuristic search-based methods. For the time being, we present some consideration as to how the current implementation of our algorithms can be sped up.

Algorithm 1 and Algorithm 3 use traditional covering array construction as a computational primitive. In particular, both algorithms compute a traditional covering array for each test cluster rather than computing a single array. Consequently, as the performance of traditional covering array generators improve, the runtime performance of Algorithm 1 and Algorithm 3 will improve proportionally. To further improve the performance, the traditonal covering arrays required by these algorithms can be computed in a parallel manner. Parallelizing Algorithm 3 is easier than parallelizing Algorithm 1. As there are no dependencies among the traditional covering arrays generated by Algorithm 3, all of these arrays can be generated in parallel. Algorithm 1, on the other hand, requires more consideration as the traditional covering arrays to be computed depend on the previously computed arrays via the seeding mechanism. One approach can be to group the test clusters and generate the required traditional covering arrays in parallel within each group. After processing a group, the current test case-aware covering array at hand can then be used as a seed for the next group. While the running time of this approach increases linearly with the number of groups rather than with the number of test clusters, the approach is likely to require more configurations than the original approach, as the test cases within a group will be less likely to share configurations. One approach to reduce the impact of this shortcoming can be to construct the groups, such that test cases that are less likely to share configurations, e.g., the ones that have conflicting constraints, are placed in the same group.

Algorithm 2, on the other hand, do not depend on traditional covering array generators. In the implementation of this algorithm, we used a sequential ASP solver to select the best row at each iteration of the algorithm (Section 4.2). One approach to speed up the computation is to use a parallel ASP solver [17], instead of a sequential one. Furthermore, in our experiments, we observed that the 2-way test case-aware covering arrays tended to cover a large percentage of the 3-pairs. For example, in the 2-way test case-aware covering arrays created by Algorithm 2, 91% of the 3-pairs (max: 94% and min: 89%) for Apache and 92% of the 3-pairs (max: 96% and min: 88%) for MySQL, on average, were already covered. Based on this observation, another approach to improve the runtime performance of Algorithm 2 can be to follow an incremental construction strategy. For instance, to compute a t-way test case-aware covering array, a $(t{-}1)$-way test case-aware covering array can be constructed first and then only those t-pairs which are not covered by the $(t{-}1)$-way array can be fed to Algorithm 2 to compute the additional rows as needed. As this approach reduces the number of t-pairs to be placed by Algorithm 2, the runtime performance of this algorithm will be likely to be improved. The same approach can also be used in a recursive manner, e.g., the $(t{-}1)$-way test case-aware covering array can be constructed from a $(t{-}2)$-way array, and so on. This approach, however, can increase the number of configurations required compared to the original approach.

Another way to reduce the construction cost is through support for efficient handling of simple, incremental changes in the configuration model. To this end, our algorithms need to be slightly modified so that they also take as input an initial seed S'. In our context, S' is a set of configuration-test cases pairs. The initial seed does not necessarily constitute a test case-aware covering array. We however assume that the seed does not violate any constraints. Given an initial seed, Algorithm 1 and Algorithm 3 need to compute an initial seed for each test case. For Algorithm 3, the initial seed S'_τ of a test case τ is the set of configurations in S', in which the test case has already been scheduled to execute. For Algorithm 1, S'_τ also contains those remaining configurations in S', which do not violate the test case-specific constraint of the test case. Once the initial seeds for the test cases are computed, line 3 in both algorithms need to change, such that the seed S_τ computed for a test case τ by the algorithms is merged with the initial seed S'_τ of the test case. Given an initial seed, Algorithm 2, on the other hand, marks all the t-pairs appearing in the seed as covered and the main loop of the algorithm is executed only for those t-pairs yet to be covered.

Once the seeding mechanism is in place, changes made to the configuration space model can be handled as follows: To add a new test case, if the test case falls in an existing test cluster, then schedule the test case with the rest of the test cases in the cluster. Otherwise (i.e., when the test case forms a new test cluster), use the current test case-aware covering array as an initial seed and execute the main loops of the algorithms only for the newly added test case. In Algorithm 2, the test case first needs to be scheduled to execute in all of its valid configurations included in the seed. To remove a test case, remove the test case from the current test case-aware covering array together with the configurations that become redundant after the removal of the test case. Changes in the test case-specific constraint of a test case can then be handled by removing the test case first and then adding the same test case with the new constraints. To add, remove, or modify a system-wide constraint, remove all the invalidated configurations (if any) and the respective test runs from the current test case-aware covering array, and use the remaining array as an initial seed.

Populating the configuration space model with a new configuration option, on the other hand, requires more consideration. In particular, we need to handle seeds with partially filled configurations. A partially filled configuration is a configuration, in which some option settings are left unset. When a new option is added, the configurations in the current test case-aware covering arrays are converted to partially filled configurations by adding the option to every configuration without a predetermined setting. Given a partially filled seed, Algorithm 2 needs to iterate over all the configuration-test cases pairs and, for each partially filled configuration, determine the "best" setting for the newly added option, given the test cases that have already been scheduled to execute in the configuration. The ASP encoding given in Figure 3 can easily be modified for this purpose. The resulting array can then be used as a fully filled initial seed. Modifying Algorithm 1 is harder. Since Algorithm 1 processes the t-pairs on a per-test case basis, making it to handle more than one test case at once is not trivial. Therefore, when adding a new configuration option, Algorithm 1 can make use of Algorithm 2 to address the change. Algorithm 3, however, handles partially filled seeds without any modifications as long as the traditonal covering array generator used in the implementation handles them. Finally, removal of a configuration option from the configuration space model can trivially be handled.

One interesting observation we make from Figure 9 is that, although Algorithm 1 and Algorithm 2 aim to minimize the number of configurations without making any attempts to reduce the number of test runs required, Algorithm 2, compared to Algorithm 1, required fewer test runs for Apache, but more test runs for MySQL (Figure 9c-d), while requiring fewer configurations than Algorithm 1 for both subject applications (Figure 9a-b). That is, for both subject applications, while Algorithm 2 covered more t-pairs per configuration on average, it covered more t-tuples per test run for Apache, but fewer t-tuples per test run for MySQL, on average.

We believe that this is mainly due to the differences between the test suites of our subject applications. First, Apache had fewer test clusters than MySQL; 17 test clusters vs. 30 test clusters (Table 3 and 4). Note that each test cluster is associated with a unique test case-specific constraint. Second, the test case-specific constraints of Apache involved fewer configuration options per constraint than those of MySQL. For example, the percentage of test case-specific constraints that involved only one option was 71 for Apache and 20 for MySQL. Therefore, when picking a configuration and the associated set of test cases, Algorithm 2 had fewer constraints, thus more choices, to consider for Apache, but more constraints, thus fewer choices, for MySQL.

We observed that in heavily constrained configuration spaces, the more the test cases are forced to share configurations, the more likely it is that fewer unique t-tuples will be covered per scheduled test run (increasing the total number of test runs required). This is because, when picking a configuration in a heavily constrained configuration space, a large portion of the configuration is likely to be dictated by the constraints of few test cases, which can reduce the number of unique t-tuples covered for the rest of the test cases that are scheduled to be executed in the same configuration. For example, when $t{=}2$ and $cop{=}30$ for Apache, Algorithm 2 covered 6.32 unique 2-tuples per test run, whereas Algorithm 1 covered 4.15 unique 2-tuples per test run, on average. In the end, Algorithm 2 required 13% fewer test runs than Algorithm 1. However, when $t{=}2$ and $cop{=}30$ for MySQL, Algorithm 2 required 7% more test runs than Algorithm 1. For this scenario, Algorithm 2 covered 2.23 unique 2-tuples per test run, whereas Algorithm 1 covered 4.08 unique 2-tuples per test run, on average. In both cases, Algorithm 2 required fewer configurations than Algorithm 1. Similar trends were observed in the rest of the experiments.

6 Threats to Validity

All empirical studies suffer from threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our studies to industrial practice.

One potential threat is that the proposed approach assumes that all test case-specific constraints are known a priori. In the presence of missing or incorrect constraints, as test cases can still skip some configurations due to unsatisfied constraints, the test case-aware covering arrays may suffer from masking effects. In such cases, the feedback driven adaptive combinational testing process we introduced in a prior work [15] can be used to iteratively detect and remove masking effects.

Another potential threat is that we have only studied two software systems; Apache and MySQL. However, both Apache and MySQL are widely-used non-trivial applications with large configuration spaces and both have been used in other related works in the literature [15], [18]. A related threat concerns the representativeness of the configuration space models and the test suites used in the experiments. Although these configuration space models and test suites were culled from the actual configuration space models and test suites of our subject applications, they only represent two sets of data points. To reduce the threats concerning the representativeness of the configuration space models, we varied the percentage of constrained options in the models (Section 5.2 and 5.3). To reduce the threats concerning the representativeness of the test suites, we studied the effect of varying the percentage of constrained test cases in the test suites (Section 5.2).

Another potential threat concerns the representativeness of the traditional covering array generator used in the experiments, namely ACTS. However, ACTS is a well-known and widely-used generator. We opted to use ACTS, since it offered the best runtime performance among the generators we experimented with.

Furthermore, our algorithms construct test case-aware covering arrays for two cost models. In one cost model, the cost of running the test cases is negligible compared to that of configuring the system and the configuration cost is the same for all configurations. In the other cost model, the cost of configuring the system is negligible compared to that of running the test cases and the execution cost is the same for all test runs. For other cost models, different algorithms may be required to minimize the cost. Furthermore, the c_c and c_r costs used in our cost function given in Section 5.3.1, are application specific. Therefore, $\hat{M}$ values may vary from one application scenario to another. To lower the strength of the dependency, we presented some consideration as to how the construction cost of test case-aware covering arrays can further be reduced (Section 5.3.3). However, we did not experiment exhaustively with these approaches and leave this as future work; the runtime performance of these approaches has yet to be studied.

Finally, we have not directly evaluated the cost-effectiveness of test case-aware covering arrays, i.e., evaluating the effectiveness, such as failure-detection capabilities, as a function of cost, such as total testing time. However, our empirical results reported in a prior work [15] strongly suggest that, as masking effects are removed, the number of failures observed and the structural code coverage obtained in testing monotonically increase.

7 Related Work

Covering arrays aim to reveal option-related failures. The results of many empirical studies strongly suggest that a majority of option-related failures in practice are caused by the interactions among only a small number of configuration options and that traditional t-way covering arrays, where t is much smaller than then the number of configuration options, are an effective and efficient way of revealing such failures [3], [9], [13], [14].

Nie et al. classify the methods for generating covering arrays, which is an NP-hard problem, into 4 main categories [26]: random search-based methods [28], heuristic search-based methods [7], [10], [12], [19], [29], mathematical methods [20], [21], [32], [33], and greedy methods [4], [6], [8], [9], [13], [23], [30], [31].

Random search-based methods employ a random selection without replacement strategy [28]. Valid configurations are randomly selected from the configuration space in an iterative manner until all the required t-tuples have been covered by the selected configurations.

Heuristic search-based methods, on the other hand, employ heuristic search techniques, such as hill climbing [12], tabu search [7], and simulated annealing [10], or AI-based search techniques, such as genetic algorithms [19] and ant colony algorithms [29]. These methods maintain a set of configurations at any given time and iteratively apply a series of transformations to the set until the set constitutes a t-way covering array.

Mathematical methods for constructing covering arrays have also been studied [21], [32], [33]. Some mathematical methods are based on recursive construction methods, which build covering arrays for larger configuration space models (i.e., the ones with a larger number of configuration options) by using covering arrays built for smaller configuration space models [21], [32]. Other mathematical methods leverage mathematical programming, such as integer programming, to compute covering arrays [33].

Greedy algorithms operate in an iterative manner [4], [6], [8], [9], [13], [23], [30], [31]. At each iteration, among the sets of configurations examined as candidates, the one that covers the most previously uncovered t-tuples is included in the covering array. The iterations terminate when all the required t-tuples have been covered.

The algorithms we present in this work fall into the category of greedy algorithms. However, while the existing greedy algorithms compute traditional covering arrays, we compute test case-aware covering arrays.

Handling system-wide inter-option constraints in the construction of traditional covering arrays have also been of interest. Cohen et al. study the nature of such constraints in configurable software systems and empirically demonstrate that ignoring such constraints leads to wasted testing efforts [11]. Mats et al. propose various techniques to efficiently handle system-wide constraints [25]. Bryce et al. introduce the concept of “soft constraints” to mark option setting combinations that are permitted, but undesirable to be included in a covering array [5].

Traditional covering arrays, while handling system-wide constraints, do not account for test case-specific constraints. In this work we, on the other hand, take test case-specific constraints into account when constructing combinatorial interaction test suites.

Seeding mechanisms in CIT approaches have been used to guarantee the inclusion of certain configurations in traditional covering arrays [9], [13], [18]. In this work, we use the seeding mechanism to construct test case-aware covering arrays.

8 Concluding remarks

The basic justification for traditional t-way covering arrays is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. In this work we hypothesize however that, in the presence of test case-specific inter-option constraints, as traditional covering arrays do not provide a systematic way of handling these constraints, many such behaviors may not be tested due to masking effects caused by the overlooked test case-specific constraints.

To evaluate this hypothesis, we conducted a series of experiments on two widely-used highly-configurable software systems, namely Apache and MySQL. We first observed that test case-specific constraints do exist in practice. Out of all the test cases we examined for our subject applications, 378 Apache test cases and 337 MySQL test cases had some test case-specific constraints. We then observed that traditional covering arrays suffered from masking effects caused by the overlooked test case-specific constraints. In a study, for example, 35% of all valid 2-way option setting combination-test case pairs (i.e., 2-pairs), on average, were not tested at all by the traditional 2-way covering arrays created. For a fixed configuration space model, higher strength covering arrays suffered relatively less compared to lower strength arrays; as t increased, t-masked percentage (i.e., the percentage of t-pairs that are masked) decreased. For a fixed value of t, as the percentage of the configuration options that are referenced by a constraint (i.e., constrained options percentage) decreased and/or the percentage of the test cases that have test case-specific constraints (i.e., constrained test cases percentage) decreased, t-masked percentage decreased. However, t-masked percentage never reached 0 in the traditional t-way covering arrays created in the experiments.

To account for test case-specific constraints and avoid harmful consequences of masking effects caused by the ignorance of such constraints, we first introduced test case-aware covering arrays. We then presented three algorithms to compute test case-aware covering arrays. A valuable observation we make is that there is often a trade-off between minimizing the number of configurations and minimizing the number of test runs in test case-aware covering arrays. Among the algorithms we introduced, Algorithm 1 and Algorithm 2 aim to minimize the number of configurations, whereas Algorithm 3 aims to minimize the number of test runs.

For the configuration space models studied in the experiments, Algorithm 2 performed better than Algorithm 1 in reducing the number of configurations required. Algorithm 2, compared to Algorithm 1, while having a higher computational complexity, significantly reduced the number of configurations by 54% when t=2 and by 45% when t=3, on average. Algorithm 3, on the other hand, performed better than Algorithm 1 and 2, in reducing the number of test runs required. Algorithm 3, compared to Algorithm 1 and 2, while requiring more configurations, reduced the number of test runs by 20% when t=2 and by 21% when t=3, on average. These results make Algorithm 3 to be of practical interest when the cost of configuring the system is negligible and Algorithm 2 to be of practical interest when the cost of running the test cases is negligible.

We also compared t-way test case-aware covering arrays with traditional t-way covering arrays. When the goal was to minimize the number of test runs, we observed that the t-way test case-aware covering arrays computed by Algorithm 3, while not suffering from any masking effects, reduced the number of test runs (except for two cases) by 11% when t=2 and by 17% when t=3, on average, compared to the traditional t-way covering arrays. When the goal was to minimize the number of configurations, however, the t-way test case-aware covering arrays expectedly required more configurations than the traditional t-way covering arrays. To evaluate the efficiency provided by t-way test case-aware covering arrays, we then compared them to traditional higher strength covering arrays that prevented all or almost all t-pairs from being masked. We observed that the test case-aware covering arrays created by Algorithm 2, while not suffering from any masking effects, reduced the number of configurations by 82% when t=2 and by 50% when t=3, on average, compared to the higher strength covering arrays.

In almost all the comparisons, constructing the test case-aware covering arrays took more time than constructing their counterparts. We, however, observed that the more the constructed arrays are reused for testing, and the more the cost of configuring the system and running the test cases, the faster test case-aware covering arrays compensate for their construction cost.

We furthermore observed that the extent to which traditional covering arrays suffer from masking effects depends on many factors, such as the strength of the covering array, the percentage of constrained options in the configuration space model, and the percentage of constrained test cases in the test suite. Therefore, we provided a guideline to the users of covering arrays to reliably estimate the consequences of masking effects without performing any system builds and test runs. This guideline can be summarized as follows: Generate a traditional t-way covering array for the scenario at hand and then compute the t-masked and t-masked percentage values. The larger the value of t-masked and/or the value of t-masked percentage, the more the covering array suffers. The evaluation results can then be used to decide if test case-aware covering arrays are required for the scenario at hand.

As future work, we plan to work on cost- and test-case aware covering arrays that support a general cost model in which the overall cost of testing can be specified at the granularity of option settings and test cases.

9 Acknowledgments

This research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme (FP7-PEOPLE-IRG-2008), and by the Scientific and Technological Research Council of Turkey (109E182).

References

[1] Advanced Combinatorial Testing System (ACTS), 2012. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html.

[2] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Cambridge, England, 2003.

[3] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[4] R. C. Bryce and C. J. Colbourn. Constructing interaction test suites with greedy algorithms. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 440–443, New York, NY, USA, 2005. ACM.

[5] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960 – 970, 2006. Advances in Model-based Testing.

[6] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing: Research articles. *Softw. Test. Verif. Reliab.*, 17:159–182, September 2007.

[7] R. C. Bryce and C. J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1082–1089, New York, NY, USA, 2007. ACM.

[8] R. C. Bryce and C. J. Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Softw. Test. Verif. Reliab.*, 19:37–53, March 2009.

[9] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–44, 1997.

[10] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 394–, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 129–139, New York, NY, USA, 2007. ACM.

[12] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[13] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proc. of the 24th Pacific Northwest Software Quality Conference*, pages 285–294, 2006.

[14] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the Int'l Conf. on Software Engineering*, pages 285–294, 1999.

[15] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 243–253, New York, NY, USA, 2011. ACM.

[16] T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Tutorial Lectures*, volume 5689 of *LNCS*, pages 40–110. Springer, 2009.

[17] E. Ellguth, M. Gebser, M. Gusowski, B. Kaufmann, R. Kaminski, S. Liske, T. Schaub, L. Schneidenbach, and B. Schnor. A simple distributed conflict-driven answer set solver. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 490–495, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] S. Fouché, M. B. Cohen, and A. Porter. Towards incremental adaptive covering arrays. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion '07, pages 557–560, New York, NY, USA, 2007. ACM.

[19] S. Ghazi and M. Ahmed. Pair-wise test coverage using genetic algorithms. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 2, pages 1420 – 1424 Vol.2, dec. 2003.

[20] A. Hartman. Software and hardware testing using combinatorial covering suites. In M. C. Golumbic and I. B.-A. Hartman, editors, *Graph Theory, Combinatorics and Algorithms*, volume 34 of *Operations Research/Computer Science Interfaces Series*, pages 237–266. Springer US, 2005.

[21] N. Kobayashi. *Design and evaluation of automatic test generation strategies for functional testing of software*. Osaka University, Osaka, Japan, 2002.

[22] D. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. on Soft. Engeering*, 30(6):418–421, 2004.

[23] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog-ipog-d: efficient test generation for multi-way combinatorial testing. *Softw. Test. Verif. Reliab.*, 18:125–148, September 2008.

[24] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, 1999.

[25] G. Mats, O. Jeff, and M. Jonas. Handling constraints in the input space when using combination strategies for software testing. Technical Report HS- IKI -TR-06-001, University of Skvde, School of Humanities and Informatics, 2006.

[26] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43:11:1–11:29, February 2011.

[27] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[28] P. J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 49–59, Washington, DC, USA, 2004. IEEE Computer Society.

[29] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '04, pages 72–77, Washington, DC, USA, 2004. IEEE Computer Society.

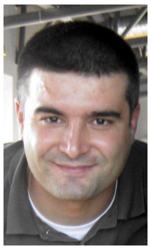
[30] K.-C. Tai and Y. Lei. A test generation strategy for pairwise testing. *Software Engineering, IEEE Transactions on*, 28(1):109 –111, jan 2002.

[31] Y.-W. Tung and W. Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431 –437 vol.1, 2000.

[32] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques*, TestCom '00, pages 59–74, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[33] A. W. Williams and R. L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, TestCom '02, pages 283–, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[34] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, Jan 2006.



Cemal Yilmaz received the BS and MS degrees in computer engineering and information science from Bilkent University, Ankara, Turkey, in 1997 and 1999, respectively. In 2005, he received the PhD degree in computer science from the University of Maryland at College Park. Between 2005 and 2008, he worked as a post-doctoral researcher at IBM Thomas J. Watson Research Center, Hawthorne, New York. He is currently an assistant professor of computer science at the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey. He received the Career Award of the Scientific and Technological Research Council of Turkey in 2009. His current research interests include software engineering and software quality assurance.

APPENDIX A
MASKING EFFECTS IN TRADITIONAL COVERING ARRAYS

Table 5 and 6 quantify the extent to which the traditional covering arrays created in the experiments suffer from masking effects caused by the overlooked test case-specific constraints. When cop=20 for Apache and MySQL, we were not able to generate traditional 5-way covering arrays because of the scalability issues we experienced with the ACTS tool.

TABLE 5: Masking effects in the traditional t-way covering arrays created for Apache.

t	option count	cop	config. count	test runs	time (mins)	2-masked	2-masked (%)	3-masked	3-masked (%)	t-masked	t-masked (%)
2	13	100%	10.30	3893.40	1	35250.30	35.1078%	n/a	n/a	35250.30	35.1078%
2	17	80%	10.50	3969.00	1	64282.70	35.2942%	n/a	n/a	64282.70	35.2942%
2	22	60%	12.00	4536.00	1	114041.40	35.8267%	n/a	n/a	114041.40	35.8267%
2	26	50%	12.30	4649.40	1	164328.10	36.1579%	n/a	n/a	164328.10	36.1579%
2	33	40%	12.30	4649.40	1	256403.10	34.1431%	n/a	n/a	256403.10	34.1431%
2	44	30%	14.00	5292.00	1	446331.30	32.6607%	n/a	n/a	446331.30	32.6607%
2	65	20%	14.10	5329.80	1	862429.60	28.2778%	n/a	n/a	862429.60	28.2778%
3	13	100%	28.20	10659.60	1	6243.00	6.2178%	132543.00	19.6027%	132543.00	19.6027%
3	17	80%	36.40	13759.20	1	8092.60	4.4432%	277826.00	16.2523%	277826.00	16.2523%
3	22	60%	42.00	15876.00	1	12012.10	3.7737%	601105.90	14.8594%	601105.90	14.8594%
3	26	50%	45.10	17047.80	1	15692.10	3.4528%	994835.10	14.2421%	994835.10	14.2421%
3	33	40%	49.10	18559.80	1	21469.00	2.8589%	2013820.00	13.3879%	2013820.00	13.3879%
3	44	30%	55.90	21130.20	1	29966.60	2.1928%	4467163.60	11.9485%	4467163.60	11.9485%
3	65	20%	62.60	23662.80	2	30498.70	1.0000%	10656026.90	8.4491%	10656026.90	8.4491%
4	13	100%	82.40	31147.20	1	2.00	0.0020%	11465.50	1.6957%	257637.70	8.3280%
4	17	80%	106.10	40105.80	1	3.00	0.0016%	23488.20	1.3740%	871431.50	7.7742%
4	22	60%	117.30	44339.40	1	7.00	0.0022%	27822.80	0.6878%	2218440.90	6.0633%
4	26	50%	137.10	51823.80	1	13.00	0.0029%	22996.70	0.3292%	3156356.20	4.0936%
4	33	40%	151.50	57267.00	4	37.50	0.0050%	47392.50	0.3151%	8403293.50	3.8445%
4	44	30%	158.00	59724.00	22	55.00	0.0040%	88987.00	0.2380%	27070689.00	3.6159%
4	65	20%	222.00	83916.00	458	0.00	0.0000%	189435.00	0.1502%	119258421.00	3.0984%
5	13	100%	199.90	75562.20	1	0.00	0.0000%	71.00	0.0105%	368286.10	3.6275%
5	17	80%	327.80	123908.40	2	0.00	0.0000%	81.00	0.0047%	1542638.10	2.8312%
5	22	60%	386.90	146248.20	5	0.00	0.0000%	276.50	0.0068%	2323385.60	0.9275%
5	26	50%	451.00	170478.00	15	0.00	0.0000%	85.00	0.0012%	12334132.90	1.8954%
5	33	40%	434.00	164052.00	112	0.00	0.0000%	262.50	0.0017%	28769495.00	1.1719%
5	44	30%	526.00	198828.00	1522	0.00	0.0000%	15.00	$\approx$ 0.0000%	132941257.00	1.1365%

TABLE 6: Masking effects in the traditional t-way covering arrays created for MySQL.

t	option count	cop	config. count	test runs	time (mins)	2-masked	2-masked (%)	3-masked	3-masked (%)	t-masked	t-masked (%)
2	12	100%	14.70	4953.90	1	20483.30	23.1900%	n/a	n/a	20483.30	23.1900%
2	15	80%	14.80	4987.60	1	42482.60	30.1342%	n/a	n/a	42482.60	30.1342%
2	20	60%	16.20	5459.40	1	57431.70	22.4616%	n/a	n/a	57431.70	22.4616%
2	24	50%	17.20	5796.40	1	64698.30	17.4051%	n/a	n/a	64698.30	17.4051%
2	30	40%	19.00	6403.00	1	125062.50	21.3342%	n/a	n/a	125062.50	21.3342%
2	40	30%	20.20	6807.40	1	199103.70	18.9347%	n/a	n/a	199103.70	18.9347%
2	60	20%	21.40	7211.80	1	334371.60	14.0106%	n/a	n/a	334371.60	14.0106%
3	12	100%	36.00	12132.00	1	3386.70	3.8342%	102780.20	17.8012%	102780.20	17.8012%
3	15	80%	37.00	12469.00	1	5566.30	3.9483%	212337.90	17.5885%	212337.90	17.5885%
3	20	60%	42.50	14322.50	1	9352.50	3.6578%	488045.10	16.0072%	488045.10	16.0072%
3	24	50%	46.60	15704.20	1	11600.70	3.1208%	783323.00	14.4284%	783323.00	14.4284%
3	30	40%	52.70	17759.90	1	14439.00	2.4631%	1334232.10	12.2239%	1334232.10	12.2239%
3	40	30%	57.20	19276.40	1	15891.80	1.5113%	2486800.60	9.3473%	2486800.60	9.3473%
3	60	20%	70.70	23825.90	2	25311.70	1.0606%	6906813.50	7.4880%	6906813.50	7.4880%
4	12	100%	108.60	36598.20	1	176.30	0.1996%	11635.70	2.0153%	264428.50	10.4976%
4	15	80%	128.10	43169.70	1	121.60	0.0863%	16098.60	1.3335%	575120.00	8.0912%
4	20	60%	145.10	48898.70	1	147.50	0.0577%	34740.90	1.1395%	1845005.50	7.1912%
4	24	50%	166.20	56009.40	1	96.30	0.0259%	41322.70	0.7611%	3128224.90	5.5248%
4	30	40%	176.67	59536.67	3	87.00	0.0148%	62573.67	0.5733%	6918783.00	4.7149%
4	40	30%	195.00	65715.00	14	121.50	0.0116%	122447.00	0.4602%	19105952.00	3.8903%
4	60	20%	221.50	74645.50	239	111.50	0.0047%	281216.50	0.3049%	79548051.00	3.0286%
5	12	100%	354.30	119399.10	1	0.20	0.0002%	201.50	0.0349%	337347.20	4.3681%
5	15	80%	395.80	133384.60	1	0.00	0.0000%	234.00	0.0194%	1235840.50	4.0557%
5	20	60%	440.00	148280.00	4	0.00	0.0000%	361.00	0.0118%	5499353.40	3.3959%
5	24	50%	470.50	158558.50	9	0.00	0.0000%	334.80	0.0062%	12398160.60	2.7627%
5	30	40%	513.50	173049.50	46	0.00	0.0000%	373.00	0.0034%	33600756.50	2.2144%
5	40	30%	560.50	188888.50	660	0.00	0.0000%	477.50	0.0018%	121823724.00	1.7279%

Appendix B
Comparing Test case-aware Covering Arrays

Table 7 and 8 provide some statistics about the test case-aware covering arrays generated in the experiments.

TABLE 7: Test case-aware covering arrays created for Apache.

algorithm	t	option count	cop	time (mins)	config. count	test runs
Algorithm 1	2	13	100%	2	48.60	3743.70
Algorithm 2	2	13	100%	1	25.50	3801.70
Algorithm 3	2	13	100%	2	131.00	3449.60
Algorithm 1	2	17	80%	2	56.80	4492.40
Algorithm 2	2	17	80%	2	27.40	4301.80
Algorithm 3	2	17	80%	2	144.20	3839.80
Algorithm 1	2	22	60%	2	65.20	5098.60
Algorithm 2	2	22	60%	21	28.67	4788.67
Algorithm 3	2	22	60%	2	171.20	4562.80
Algorithm 1	2	26	50%	3	69.40	5514.00
Algorithm 2	2	26	50%	55	30.00	5112.33
Algorithm 3	2	26	50%	2	172.20	4551.60
Algorithm 1	2	33	40%	3	77.40	6234.60
Algorithm 2	2	33	40%	81	32.50	5688.00
Algorithm 3	2	33	40%	3	172.80	4587.80
Algorithm 1	2	44	30%	4	84.40	7211.40
Algorithm 2	2	44	30%	106	34.00	6256.50
Algorithm 3	2	44	30%	3	201.40	5301.20
Algorithm 1	2	65	20%	7	98.40	8060.00
Algorithm 2	2	65	20%	133	37.50	7512.50
Algorithm 3	2	65	20%	4	202.60	5305.80
Algorithm 1	3	13	100%	3	114.30	9512.10
Algorithm 2	3	13	100%	7	65.90	8922.80
Algorithm 3	3	13	100%	2	349.90	8173.00
Algorithm 1	3	17	80%	5	141.40	11655.80
Algorithm 2	3	17	80%	10	78.60	10953.20
Algorithm 3	3	17	80%	4	422.00	9775.40
Algorithm 1	3	22	60%	9	165.40	14376.20
Algorithm 2	3	22	60%	99	86.00	12368.33
Algorithm 3	3	22	60%	6	506.20	11816.00
Algorithm 1	3	26	50%	14	171.80	15917.80
Algorithm 2	3	26	50%	179	91.00	13862.67
Algorithm 3	3	26	50%	9	549.40	12414.00
Algorithm 1	3	33	40%	30	188.60	18447.40
Algorithm 2	3	33	40%	287	103.00	16236.00
Algorithm 3	3	33	40%	18	592.60	13924.40
Algorithm 1	3	44	30%	77	214.00	21087.40
Algorithm 2	3	44	30%	491	113.50	19204.00
Algorithm 3	3	44	30%	45	692.80	15558.20
Algorithm 1	3	65	20%	291	247.00	24459.00
Algorithm 2	3	65	20%	1476	131.00	23269.50
Algorithm 3	3	65	20%	176	812.20	18359.40

TABLE 8: Test case-aware covering arrays created for MySQL.

algorithm	t	option count	cop	time (mins)	config. count	test runs
Algorithm 1	2	12	100%	3	73.10	4617.20
Algorithm 2	2	12	100%	1	42.20	5112.70
Algorithm 3	2	12	100%	3	225.60	3609.10
Algorithm 1	2	15	80%	3	81.00	5100.60
Algorithm 2	2	15	80%	2	43.10	5727.10
Algorithm 3	2	15	80%	3	295.00	4268.20
Algorithm 1	2	20	60%	4	91.60	5599.80
Algorithm 2	2	20	60%	17	46.33	6351.00
Algorithm 3	2	20	60%	3	352.20	4825.40
Algorithm 1	2	24	50%	5	96.80	6008.80
Algorithm 2	2	24	50%	49	44.33	6721.00
Algorithm 3	2	24	50%	4	365.40	5059.40
Algorithm 1	2	30	40%	6	103.40	6531.20
Algorithm 2	2	30	40%	83	46.00	7249.33
Algorithm 3	2	30	40%	4	391.60	5411.20
Algorithm 1	2	40	30%	8	112.40	7254.60
Algorithm 2	2	40	30%	108	49.67	7773.67
Algorithm 3	2	40	30%	5	416.40	5577.80
Algorithm 1	2	60	20%	14	120.00	8010.40
Algorithm 2	2	60	20%	140	50.50	8824.00
Algorithm 3	2	60	20%	8	453.40	5918.40
Algorithm 1	3	12	100%	5	203.90	12419.00
Algorithm 2	3	12	100%	17	130.00	14726.33
Algorithm 3	3	12	100%	4	576.30	10392.50
Algorithm 1	3	15	80%	9	230.20	14428.60
Algorithm 2	3	15	80%	66	144.40	15962.40
Algorithm 3	3	15	80%	6	873.00	11831.20
Algorithm 1	3	20	60%	18	260.80	16954.20
Algorithm 2	3	20	60%	327	152.00	17940.33
Algorithm 3	3	20	60%	10	1068.00	13799.60
Algorithm 1	3	24	50%	31	282.60	18198.80
Algorithm 2	3	24	50%	665	155.67	20082.00
Algorithm 3	3	24	50%	16	1173.40	14772.60
Algorithm 1	3	30	40%	64	308.40	20027.40
Algorithm 2	3	30	40%	1504	166.67	22850.33
Algorithm 3	3	30	40%	28	1299.20	16119.00
Algorithm 1	3	40	30%	167	342.40	22785.40
Algorithm 2	3	40	30%	4599	181.00	26062.00
Algorithm 3	3	40	30%	71	1464.60	18253.40
Algorithm 1	3	60	20%	621	384.67	25584.67
Algorithm 2	3	60	20%	25277	195.00	30169.00
Algorithm 3	3	60	20%	271	1713.00	21087.00