Sabancı Üniversitesi

Seer: A Lightweight Online Failure Prediction Approach

Cemal Yilmaz

Faculty of Engineering and Natural Sciences

Sabanci University

Istanbul, Turkey

cyilmaz@sabanciuniv.edu

Outline

Part I

- Seer: A lightweight online failure prediction approach

Part II

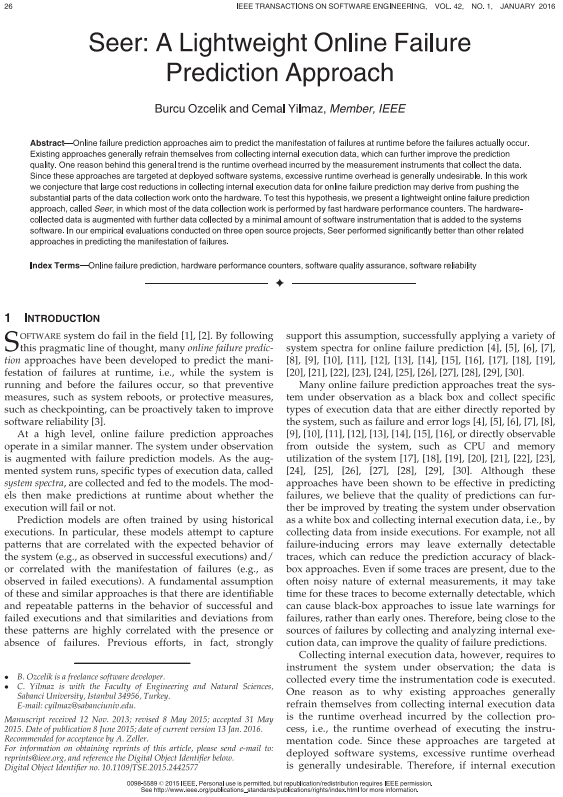
- SpyDetector: An approach for detecting side-channel attacks at runtime

Part III

- Making combinatorial interaction testing more practical

Seer: A lightweight online failure prediction approach

Burcu Ozcelik
M.S., 2012



26 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 42, NO. 1, JANUARY 2016

Seer: A Lightweight Online Failure Prediction Approach

Burcu Ozcelik and Cemal Yilmaz, *Member, IEEE*

Abstract—Online failure prediction approaches aim to predict the manifestation of failures at runtime before the failures actually occur. Existing approaches generally refrain themselves from collecting internal execution data, which can further improve the prediction quality. One reason behind this general trend is the runtime overhead incurred by the measurement instruments that collect the data. Since these approaches are targeted at deployed software systems, excessive runtime overhead is generally undesirable. In this work we conjecture that large cost reductions in collecting internal execution data for online failure prediction may derive from pushing the substantial parts of the data collection work onto the hardware. To test this hypothesis, we present a lightweight online failure prediction approach, called *Seer*, in which most of the data collection work is performed by fast hardware performance counters. The hardware-collected data is augmented with further data collected by a minimal amount of software instrumentation that is added to the systems software. In our empirical evaluations conducted on three open source projects, Seer performed significantly better than other related approaches in predicting the manifestation of failures.

Index Terms—Online failure prediction, hardware performance counters, software quality assurance, software reliability

1 Introduction

Software system do fail in the field [1], [2]. By following this pragmatic line of thought, many *online failure prediction* approaches have been developed to predict the manifestation of failures at runtime, i.e., while the system is running and before the failures occur, so that preventive measures, such as system reboots, or protective measures, such as checkpointing, can be proactively taken to improve software reliability [3].

At a high level, online failure prediction approaches operate in a similar manner. The system under observation is augmented with failure prediction models. As the augmented system runs, specific types of execution data, called *system spectra*, are collected and fed to the models. The models then make predictions at runtime about whether the execution will fail or not.

Prediction models are often trained by using historical executions. In particular, these models attempt to capture patterns that are correlated with the expected behavior of the system (e.g., as observed in successful executions) and/or correlated with the manifestation of failures (e.g., as observed in failed executions). A fundamental assumption of these and similar approaches is that there are identifiable and repeatable patterns in the behavior of successful and failed executions and that similarities and deviations from these patterns are highly correlated with the presence or absence of failures. Previous efforts, in fact, strongly support this assumption, successfully applying a variety of system spectra for online failure prediction [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30].

Many online failure prediction approaches treat the system under observation as a black box and collect specific types of execution data that are either directly reported by the system, such as failure and error logs [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], or directly observable from outside the system, such as CPU and memory utilization of the system [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30]. Although these approaches have been shown to be effective in predicting failures, we believe that the quality of predictions can further be improved by treating the system under observation as a white box and collecting internal execution data, i.e., by collecting data from inside executions. For example, not all failure-inducing errors may leave externally detectable traces, which can reduce the prediction accuracy of black-box approaches. Even if some traces are present, due to the often noisy nature of external measurements, it may take time for these traces to become externally detectable, which can cause black-box approaches to issue late warnings for failures, rather than early ones. Therefore, being close to the sources of failures by collecting and analyzing internal execution data, can improve the quality of failure predictions.

Collecting internal execution data, however, requires to instrument the system under observation; the data is collected every time the instrumentation code is executed. One reason as to why existing approaches generally refrain themselves from collecting internal execution data is the runtime overhead incurred by the collection process, i.e., the runtime overhead of executing the instrumentation code. Since these approaches are targeted at deployed software systems, excessive runtime overhead is generally undesirable. Therefore, if internal execution

- *B. Ozcelik is a freelance software developer.*
- *C. Yilmaz is with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul 34956, Turkey. E-mail: cyilmaz@sabanciuniv.edu.*

Manuscript received 12 Nov. 2013; revised 8 May 2015; accepted 31 May 2015. Date of publication 8 June 2015; date of current version 13 Jan. 2016. Recommended for acceptance by A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2442577

Funded by:
1. Scientific and Technological Research Council of Turkey (109E182)
2. 7th European Community Framework Programme (PIRG04-GA-2008-239484)

Online failure prediction

Software systems do fail in the field

To improve software reliability, online failure prediction approaches aim to predict the manifestation of failures at runtime before they actually occur so that preventive and protective measures can be proactively taken

Online failure prediction approaches

At a high level, many online failure prediction approaches operate in a similar manner

- The system under observation is instrumented with failure prediction models
- As the instrumented system runs, specific types of data, called system spectra, are collected and fed to the models
- The models then make predictions at runtime about whether the execution will fail or not

Developing prediction models

Prediction models are often trained by using historical executions

These models typically attempt to capture patterns that are correlated with the expected behavior of the system (e.g., as observed in successful executions) and/or correlated with the manifestation of failures (e.g., as observed in failed executions)

Fundamental assumption

There are identifiable and repeatable patterns in the behavior of failing and passing executions and similarities and/or deviations from these patterns are often highly correlated with the presence or absence of failures

Caveat

Correlation does not mean causation!

However, many many studies in the literature strongly suggest that even rough approximations obtained by using correlations are generally of great practical importance in many related application domains, such as

- Fault detection
- Fault characterization
- Fault localization
- Static failure prediction
- Software repository mining
- Other types of data-driven program analysis
- …

Existing online failure prediction approaches

- **Many treat the system under observation as a black box**
- **Collect specific types of execution data that are either directly reported by the system, such as failure and error logs, or directly observable from outside the system, such as CPU and memory utilization of the system**
- **One reason as to why existing approaches generally refrain themselves from collecting internal execution data is the runtime overhead incurred by the collection process**
- **Since these approaches are targeted at deployed software systems, excessive runtime overhead is generally undesirable**

Hypothesis I

We conjecture that the quality of predictions can further be improved by treating the system under observation as a white box and collecting internal execution data, i.e., by collecting data from inside executions

For example

- Not all failure-inducing errors may leave externally detectable traces, which can reduce the prediction accuracy of black-box approaches
- Even if some traces are present, due to the often noisy nature of external measurements, it may take time for these traces to become externally detectable, which can cause black-box approaches to issue late warnings for failures, rather than early ones

What about runtime overhead???

To reduce the runtime overhead, we push the substantial parts of the data collection work onto the hardware

In particular, much of the data is collected by fast hardware performance counters, which is then augmented with further data collected by a minimal amount of software instrumentation that is added to the systems software

We call this type of spectra hybrid spectra

Hardware performance counters

CPU resident counters that record various low level events occurring on a CPU

- E.g., number of machine instructions executed, the number of branches taken, the number of cache hits/misses experienced, etc.

Today's general-purpose CPUs include a fair number of such counters

To activate these counters, programs issue instructions indicating the type of event to be counted and the physical counter to be used

Once activated, counters count the events of interest and store the counts in a set of special purpose registers

These registers can also be read and reset programmatically at runtime

Hypothesis II

We conjecture that hardware performance counters can be used to collect data from inside executions in an unobtrusive manner, and that the data collected can be used for online failure prediction, all at acceptable runtime overhead costs

But, abstracting program executions is genuinely difficult

Sabancı Üniversitesi

A program execution is a complex event

- Control flow
- Data flow
- Aliases
- Side effects
- Non-determinism
- Interactions with hardware and software platforms
- …

It is often unclear

- What to collect
- How to analyze
- How to model

Time will tell: Fault localization using time spectra

music of a function

Leverages time spectra as an abstraction mechanism for program executions

- time spectra ≈ traces of method execution times

Identifies and scores methods that take a "suspicious" amount of time to execute

- In a sense, time tells us everything, yet nothing in particular

Different than performance debugging where the goal is to detect hotspots in the program code

Cemal Yilmaz, A. Paradkar, and C. Williams, "Time will tell: fault localization using time spectra." In the *Proceedings of the 30th International Conference on Software Engineering* (ICSE '08). pp. 81-90, ACM, New York, NY, USA, 2008.

Combining hardware and software instrumentation for fault detection

Sabancı Üniversitesi

An approach for fault detection

Analyzes hybrid spectra in an offline manner to detect failures after they have occurred, i.e., after the executions have terminated

Identifies patterns in unsuccessful executions and checks to see if a previously unseen execution exhibits behaviors that are similar to these likely-to-be-failure-inducing patterns

Cemal Yilmaz and Adam Porter, "Combining hardware and software instrumentation to classify program executions." *In the Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, pp. 67-76, ACM, New York, NY, USA, 2010.

Seer

Analyzes hybrid spectra in an online manner to predict the manifestation of failures before they actually occur

Consists of two phases

- An offline training phase
  - —Takes as input a set of passing and failing executions
  - —Produces as output a set of prediction models
- An online monitoring phase
  - —The system is first augmented with the prediction models and then deployed
  - —The models are then used at runtime to make binary predictions about the future of the executions, i.e., passing or failing

Some challenges

Hardware performance counters do not distinguish between the instructions issued by different processes

- Solution: Use virtual hardware counters that can track hardware events on a per-process basis

Hardware performance counters have limited visibility into the programs being executed, e.g., by themselves they do not know, for example, to which program function the current instruction belongs

- Solution: We chose to associate the hardware-collected data with function invocations

Collecting hybrid spectra in a nutshell

Activate a hardware performance counter at the beginning of an execution

Read the value of the counter before and after each function invocation of interest and attribute the difference to the invocation

Further itemize the event count to reflect the number of events occurred in the body of the function and in each callee

Deactivate the counter at the end of the execution

Hybrid spectra: Example

An example hybrid spectra collected for a program function by counting the number of machine instructions executed

body	f66	f120	…	status
8650	4511	779725	…	P
9429	4512	779724	…	P
10783	-1	779994	…	P
9426	4511	779725	…	F
9780	-1	779993	…	F
…	…	…	…	…

Counter overhead

Hardware performance counters are quite fast

Using a 2GHz Dual Intel Xeon machine running CentOS 5.2 OS with 2GB of RAM

- The overhead of running a system with counters activated: Virtually free
- The overhead of reading the value of a virtual counter: ≈ 45 clock cycles

However, reading them before and after every function invocation may still be expensive depending on the number of invocations occurring

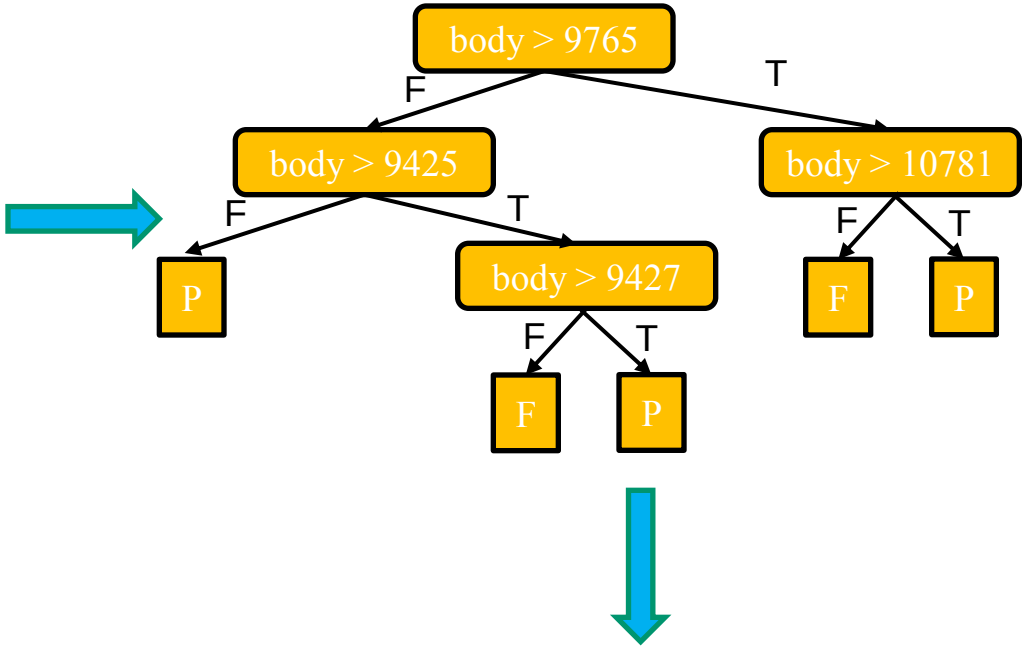
Frequency filtering

Filters out functions that are invoked more than a cutoff value (in our case cutoff=50)

Seer functions

Functions that best distinguish failing executions from passing executions

body	f66	f120	…	status
8650	4511	779725	…	P
9429	4512	779724	…	P
10783	-1	779994	…	P
9426	4511	779725	…	F
9780	-1	779993	…	F
…	…	…	…	…

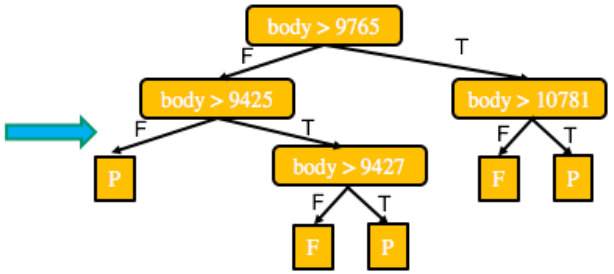


If F-measure is greater than a cutoff, then mark as a seer function (in our case cutoff=0.8, 0.9, 0.95)

Monitoring phase

The seer functions are instrumented and the system is deployed system, such that after every execution of a seer function a binary prediction, i.e., (P)assing or (F)ailing, is made about the future of the execution

body	f66	f120	...	status
8650	4511	779725	...	P
9429	4512	779724	...	P
10783	-1	779994	...	P
9426	4511	779725	...	F
9780	-1	779993	...	F
...	...	...	...	...



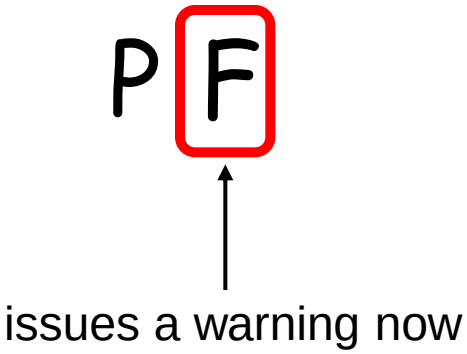

```
if (body > 9765){
   if (body > 10781) return P
   else return F
}
else {
   if (body > 9425){
      if (body > 9427) return P
      else return F
    }
    else return P
}
```

Health index

Is a string of P and F characters representing the predictions made by seer functions

Point-wise prediction approach

A warning for a possible failure is issued after seen the first F prediction from a seer function

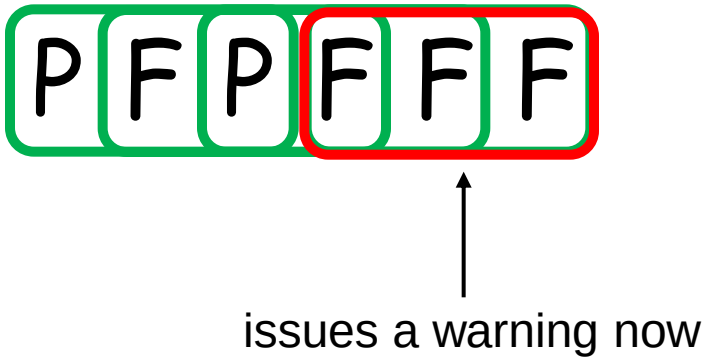


Sliding window prediction approach

Uses fixed-length sliding windows of seer predictions with a gap of one prediction

Assigns a score to each window, which grows linearly with the number of F predictions in the window

Issues a warning when the score gets larger than a cutoff value



Experiments

Subject applications: grep, flex, and sed

Hybrid spectra used

- TOT_INS: The number of machine instructions
- BRN_TKN: The number of branches
- LST_INS: The number load and store instructions

Software spectra used

- Call: Keeps track of the functions invoked
- Visit: Keeps track of the frequency of invocations

Hybrid vs software spectra

Hybrid spectra

body	f66	f120	…	status
8650	4511	779725	…	P
9429	4512	779724	…	P
10783	-1	779994	…	P
9426	4511	779725	…	F
9780	-1	779993	…	F
…	…	…	…	…

Call spectra

body	f66	f120	…	status
1	1	1	…	P
1	1	1	…	P
1	-1	1	…	P
1	1	1	…	F
1	-1	1	…	F
…	…	…	…	…

Visit spectra

body	f66	f120	…	status
1	1	11	…	P
1	2	11	…	P
1	-1	12	…	P
1	3	11	…	F
1	-1	11	…	F
…	…	…	…	…

Evaluation framework

F-measure

- What was the prediction accuracy?

Warning time

- How early we predicted the failures?

Runtime overheads

- What was the runtime overhead?

Synopsis of results

At the lowest level of runtime overheads attained, Seer, on average, predicted the failures

- about **54%** way through the executions;
- with an F- measure of **0.77**; and
- a runtime overhead of **1.98%**

At the highest level of prediction accuracies attained, Seer, on average, predicted the failures

- about **56%** way through the executions;
- with an F-measure of **0.88**; and
- a runtime overhead of **2.67%**

Duration of an execution is measured as the number of function calls made in the execution

Synopsis of results

Hybrid spectra provided significantly more accurate predictions than software spectra

Sliding windows prediction approach was significantly better than the point-wise prediction approach

Seer performed slightly better in the presence of multiple faults, compared to the presence of a single fault

Seer performed significantly better than fault screeners - an alternative state-of-the-art failure prediction approach

Concluding remarks

The empirical results support our basic hypothesis that hardware performance counters can be used to collect data from inside executions in an unobtrusive manner, and that the data collected can be used for online failure prediction, all at acceptable runtime overheads

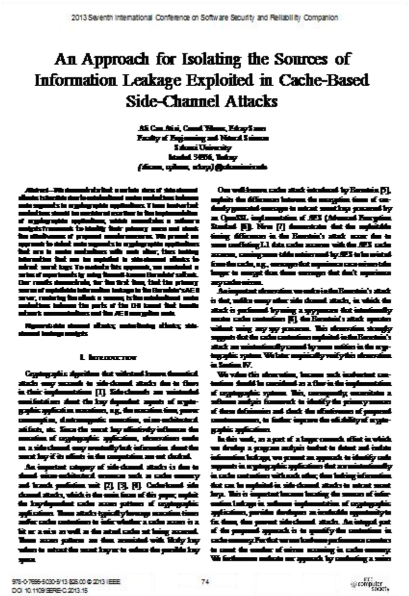
Future work

Combining low-level internal execution data collected by hardware performance counters with high-level external execution data for online failure prediction

Designing hardware components for hardware-accelerated dynamic program analysis

Detecting side-channel attacks at runtime

Yusuf Kulah
M.S., 2014


2013 Seventh International Conference on Software Security and Reliability Companion

An Approach for Isolating the Sources of Information Leakage Exploited in Cache-Based Side-Channel Attacks


Advance Access publication on 23 April 2015

© The British Computer Society 2015. All rights reserved.
For Permissions, please email: journals.permissions@oup.com
doi:10.1093/comjnl/bxv027

A Generic Method for the Analysis of a Class of Cache Attacks: A Case Study for AES

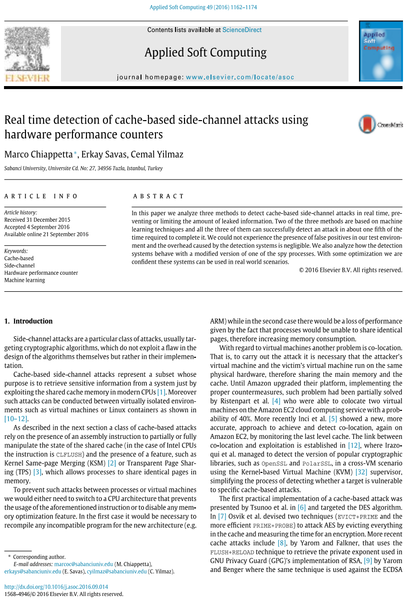
Erkay Savaş* and Cemal Yilmaz

Faculty of Engineering and Natural Sciences, Sabancı University, İstanbul, Turkey
*Corresponding author: erkays@sabanciuniv.edu

Keywords: cache attacks; timing attacks; side-channel attacks
Received 9 October 2014; revised 25 March 2015
Handling editor: Ing-Ray Chen

1. INTRODUCTION

Section D: Security in Computer Systems and Networks
The Computer Journal, Vol. 58 No. 10, 2015


Applied Soft Computing 49 (2016) 1162–1174

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Real time detection of cache-based side-channel attacks using hardware performance counters

Marco Chiappetta*, Erkay Savas, Cemal Yilmaz

Sabanci University, Universite Cd. No: 27, 34956 Tuzla, Istanbul, Turkey

1. Introduction

Funded by Scientific and Technological Research Council of Turkey (110E037)

Side-channel attacks

Side-channels are unintended manifestations about the key dependent aspects of cryptographic application executions

- E.g., the execution time, power consumption, electromagnetic emanation, micro-architectural artifacts, etc.

Since the secret key effectively influences the execution of cryptographic applications, observations made on a side-channel may eventually leak information about the secret key if its effects in the computation are not cloaked

Cache-based side channel attacks

Exploit the key-dependent cache access patterns of cryptographic applications to reveal the secret keys processed by cryptographic applications

Many of these attacks use a spy process to intentionally create cache contentions with the cryptographic application

The contentions are then analyzed to infer cache access patterns, which are in turn associated with likely key values to extract the secret key processed by the cryptographic application or to reduce the possible key space

Types of attacks

- Prime-and-probe attacks
- Flush-and-reload attacks
- …

Advanced encryption standard (AES)

AES is a symmetric-key block cipher algorithm established as a standard by NIST

It comes in different flavors

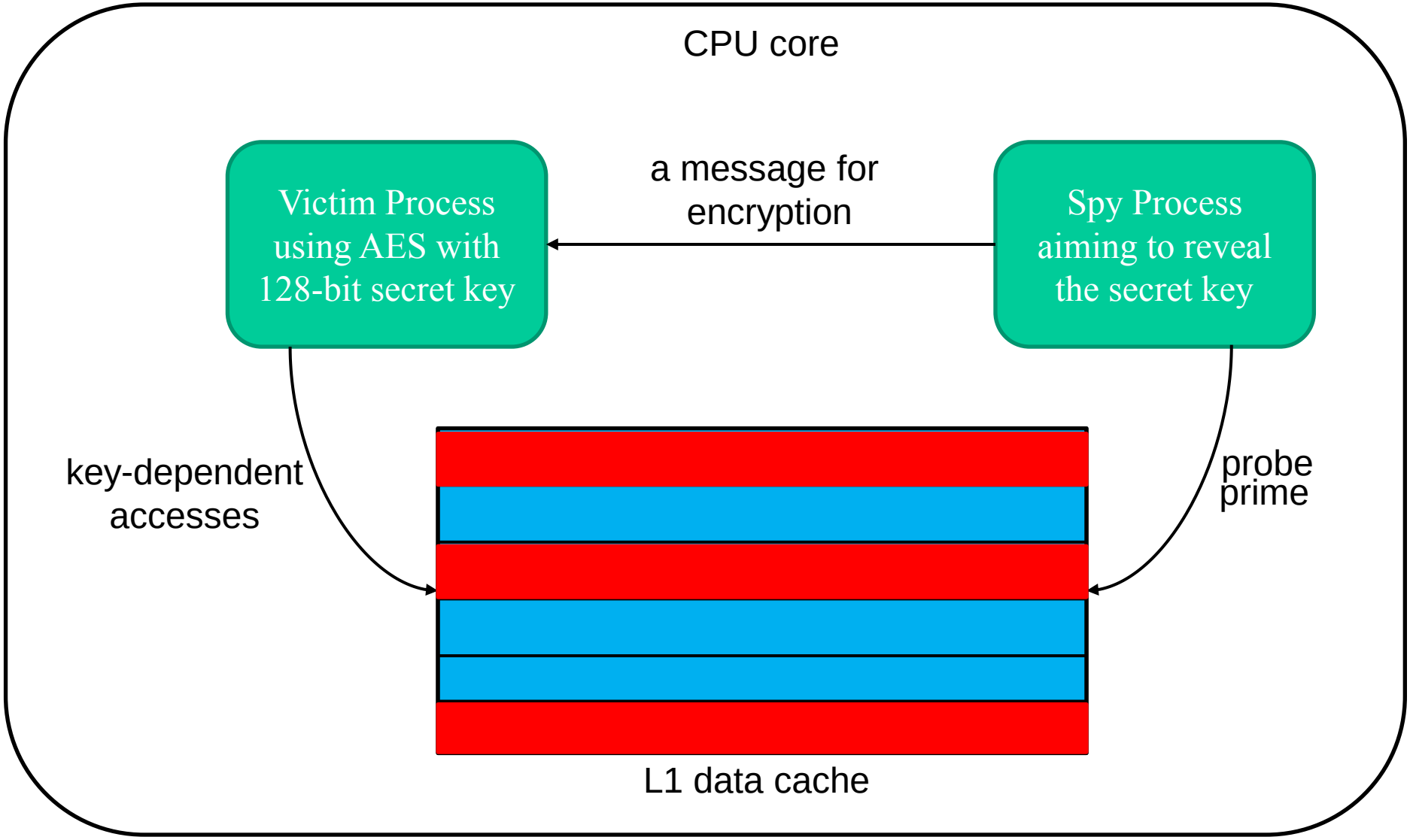
The one, which is of interest in this study, encrypts (decrypts) a plaintext (ciphertext) in blocks of 128 bits using a 128-bit secret key

To speed up processing, five static lookup tables, each of which has 256 32-bit entries (5 KB in total), are employed

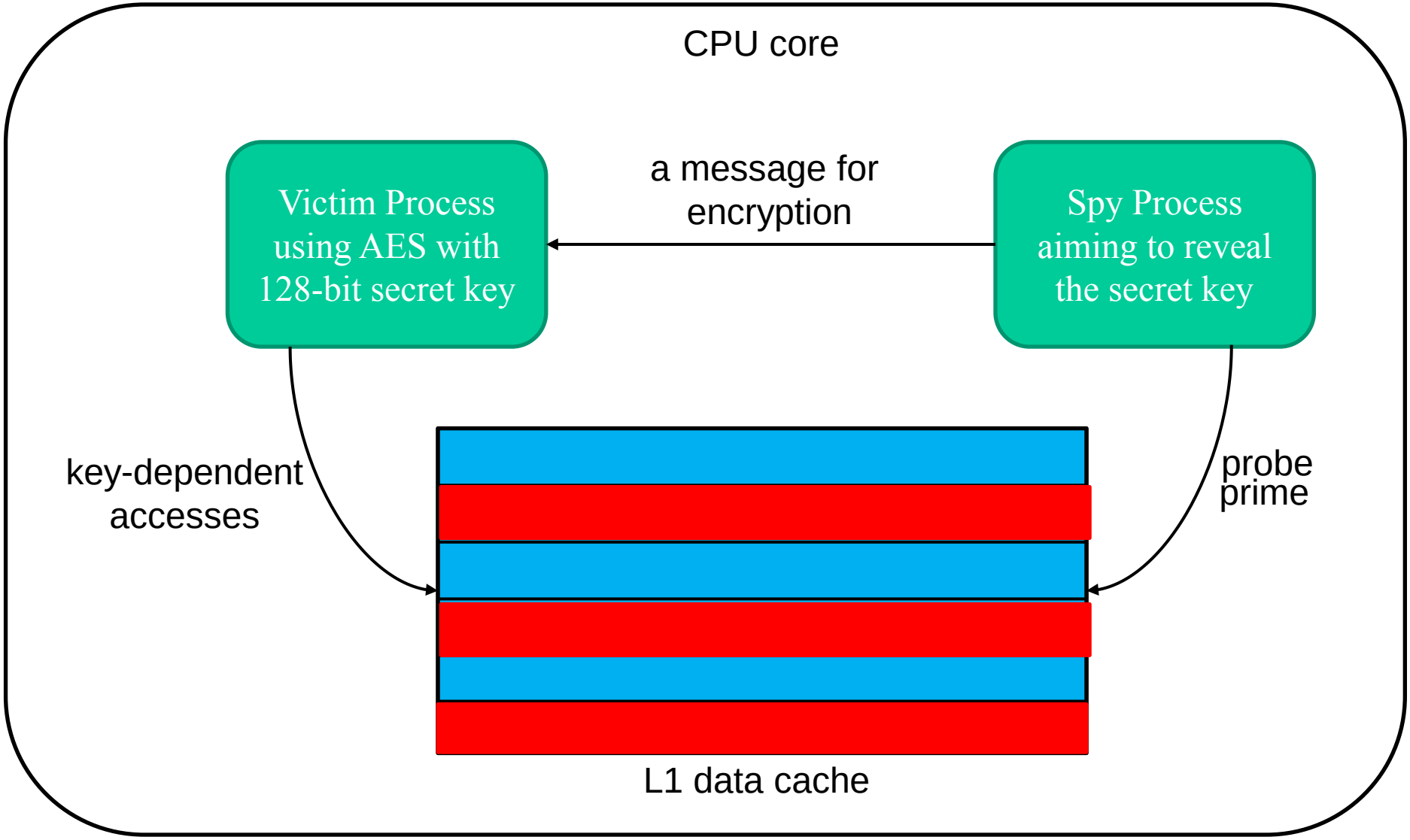
Table lookups are key dependent

160 table lookups are performed while processing a 128-bit input block

Prime-and-probe attacks on AES



Prime-and-probe attacks on AES



Countermeasures

Have been so far taken retrospectively

- First, a new attack method emerges
- Then, the attack is analyzed and countermeasures to prevent the same or similar types of attacks are developed
- Finally, the countermeasures are deployed

However, spy processes seem to have been quite successfully adapted themselves to such retrospective countermeasures

Proposed approach

...

Sabancı Üniversitesi

Making combinatorial interaction testing more practical

A motivating example: MySQL

MySQL characteristics

- Database management system
- Large user community – 6M+
- Large code base – ≈1M
- Geographically distributed developers
- Continuous evolution – 200+ commits per month

A highly configurable system

- 100+ configuration options
- Dozens of OS, compiler, and platform combinations

A motivating example: MySQL

100+ configuration options

Assuming each option has two levels of settings

- 2^{100+} configurations for testing

Assuming each configuration takes 1 second to test

- 2^{100+} secs ≃ 10^{20+} centuries for exhaustive testing
- [illegible] ago!

Which configurations should be tested?

Exhaustive testing is impossible!

Configuration space model

In its simplest form, the model includes

- a set of configuration options $O=\{o_1, o_2, \ldots, o_n\}$
- their possible settings $V=\{V_1, V_2, \ldots, V_k\}$

Implicitly defines the configuration space

```
[Options]
o1: {0, 1, 2}
o2: {0, 1, 2}
o3: {0, 1, 2}

[Constraints]
# none, for now!
```

t-way covering arrays

Given a model, a t-way covering array is a set of configurations, in which each possible combination of option settings for every combination of *t* options appears at least once

***t*: coverage strength**

o1	o2	o3
0	0	0
0	1	1
0	2	2
1	0	1
1	1	2
1	2	0
2	0	2
2	1	0
2	2	1

A 2-way covering array

Basic justification

***t*-way covering arrays can efficiently exercise all system behaviors caused by the settings of *t* or fewer options**

Example covering array sizes

coverage strength (t)	# of binary options (n)	covering array size
3	12	15
3	26	22
3	68	31
3	256	46
4	12	24
4	26	51
4	67	67
4	256	160

Applications to software testing

Input parameter testing

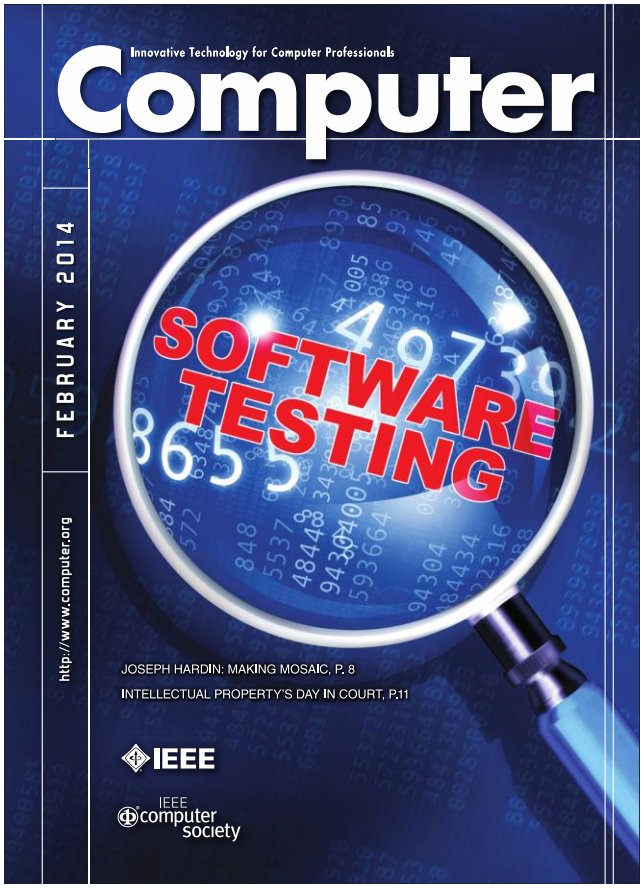
Configuration testing

GUI testing

Testing of software product lines

…

Moving forward with combinatorial interaction testing



COVER FEATURE

Moving Forward with Combinatorial Interaction Testing

Cemal Yilmaz, *Sabanci University, Istanbul, Turkey*
Sandro Fouché, *Towson University*
Myra B. Cohen, *University of Nebraska-Lincoln*
Adam Porter, *University of Maryland at College Park*
Gulsen Demiroz and Ugur Koc, *Sabanci University, Istanbul, Turkey*

Combinatorial interaction testing (CIT) is an effective failure detection method for many types of software systems. This review discusses the current approaches CIT uses in detecting parameter interactions, the difficulties of applying it in practice, recent advances, and opportunities for future research.

Modern software systems frequently offer hundreds or even thousands of configuration options. A recent version of the Apache webserver, for example, has 172 user-configurable options—158 of these are binary, eight are ternary, four have four settings, one has five, and the last one has six. Consequently, this system has 1.8×10^{55} unique configurations. The implications for fully testing such a system are clear: it cannot be done. Even if it only took one second to test each configuration, the time needed to test all possible system configurations is longer than the Earth has existed. Equally astounding calculations emerge when looking at other kinds of system variability that also require testing, such as user inputs, sequences of operations, or protocol options.

For this reason, the testing of industrial systems will almost always involve sampling enormous input spaces and testing representative instances of a system's behavior. In practice, this sampling is commonly performed with techniques collectively referred to as combinatorial interaction testing (CIT).[1,2] CIT typically models a system under test (SUT) as a set of factors (choice points or parameters), each of which takes its values from a particular domain. Based on this model, CIT then generates a sample that meets specified coverage criteria; that is, the sample contains specified combinations of the factors and their values. For instance, pairwise testing requires that each possible combination of values, for each pair of factors, appears at least once in the sample. This is the most common case.[3]

0018-9162/14/$31.00 © 2014 IEEE Published by the IEEE Computer Society

Masking effects in combinatorial testing

2011 International Symposium on Software Testing and Analysis (ISSTA 2011)

Proceedings

Matthew B. Dwyer and Frank Tip

July 17–21, 2011
Toronto, ON, Canada

Feedback Driven Adaptive Combinatorial Testing

Emine Dumlu and Cemal Yilmaz
Faculty of Eng. and Nat. Sci.
Sabanci University
Istanbul, Turkey
{edumlu,cyilmaz}@sabanciuniv.edu

Myra B. Cohen
Dept. of Comp. Sci. & Eng.
University of Nebraska-Lincoln
Lincoln, NE 68558
myra@cse.unl.edu

Adam Porter
Dept. of Comp. Sci.
University of Maryland
College Park, MD 20742
aporter@cs.umd.edu

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. The basic justification for CIT approaches is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. We conjecture, however, that in practice many such behaviors are not actually tested because of *masking effects* – failures that perturb execution so as to prevent some behaviors from being exercised. In this work we present a feedback-driven, adaptive, combinatorial testing approach aimed at detecting and working around masking effects. At each iteration we detect potential masking effects, heuristically isolate their likely causes, and then generate new covering arrays that allow previously masked combinations to be tested in the subsequent iteration. We empirically assess the effectiveness of the proposed approach on two large widely used open source software systems. Our results suggest that masking effects do exist and that our approach provides a promising and efficient way to work around them.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Combinatorial testing, adaptive testing, covering arrays, software quality assurance

1. INTRODUCTION

Software customization, through the modification of runtime or compile-time preferences, allows users to make controlled variations to how their software behaves. Customizable systems such as web servers (e.g. Apache), databases (e.g. MySQL), application servers (e.g. Tomcat) or office applications (e.g. MS Word) which have dozens or even hundreds of customizable options can have an enormous number of configurations.

While validating the correctness of the system across its entire configuration space is desirable, exhaustive testing of all configurations is generally infeasible. One solution approach, called combinatorial interaction testing (CIT), systematically samples the configuration space and tests only the selected configurations [2, 7, 9, 11, 18].

CIT approaches generally work by first defining a model of the system's configuration space – the set of valid ways it can be configured. Typically, this model includes a set of configuration options, each of which can take on a small number of option settings. Given this model, CIT methods next compute a small set of concrete configurations, a t-way covering array, in which each possible combination of option settings for every combination of t options appears at least once [7]. Finally, the system is tested by running its test suite on each configuration in the covering array.

Covering array approaches generally assume that there are no unknown control dependencies among the configuration options, option setting combinations that effectively cancel other options setting combinations. Known control dependencies are worked around by specifying constraints [7, 8] or by defining a set of default test cases in addition to the covering array [7]. Given these assumptions, and assuming the existence of a well constructed test suite, the basic justification for 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 practice many such behaviors are not actually tested due to *masking effects*. That is, we believe that some test failures can perturb program execution in ways that prevent other behaviors from being tested. Moreover, we believe that masking effects are not accounted for with current test processes. As a result, developers may develop a 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 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

ISSTA'11, July 17–21, 2011, Toronto, ON, Canada

Test-case aware combinatorial testing

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 interoption constraints into account, do not provide a systematic way of handling test case-specific interoption 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 paper, we hypothesize, however, that in the presence of test case-specific interoption 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 ignorance of 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 interoption 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].

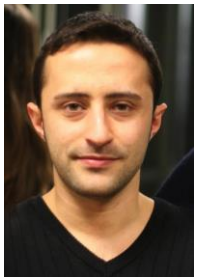
Once a covering array is generated, the system is then tested by running its test cases in all the selected configurations. By doing so, traditional covering arrays assume that all test cases can run in all the selected configurations. In this paper, we argue that the test cases of configurable systems 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 3,789 Apache test cases and 337 out of 738 MySQL test cases had test-case specific interoption 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 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

• *The author is with the Faculty of Engineering and Natural Sciences, Sabanci University, Tuzla, Istanbul 34956, Turkey. E-mail: cyilmaz@sabanciuniv.edu.*

Manuscript received 26 Mar. 2012; revised 14 July 2012; accepted 13 Sept. 2012; published online 21 Sept. 2012.
Recommended for acceptance by J. Grundy.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-03-0071.
Digital Object Identifier no. 10.1109/TSE.2012.65.

 Published by the IEEE Computer Society

Computing test case-aware covering arrays



Ugur Koc
M.S., 2014

...

Feedback driven adaptive combinatorial testing

Reducing Masking Effects in Combinatorial Interaction Testing: A Feedback Driven Adaptive Approach

Cemal Yilmaz, *Member, IEEE*, Emine Dumlu, Myra B. Cohen, *Member, IEEE*, and Adam Porter, *Senior Member, IEEE*

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. The basic justification for CIT approaches is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. We conjecture, however, that in practice some of these behaviors are not actually tested because of *unanticipated masking effects* – test case failures that perturb system execution so as to prevent some behaviors from being exercised. While prior research has identified this problem, most solutions require knowing the masking effects a priori. In practice this is impractical, if not impossible. In this work, we reduce the harmful consequences of masking effects. First we define a novel interaction testing criterion, which aims to ensure that each test case has a fair chance to test all valid t-way combinations of option settings. We then introduce a feedback driven adaptive combinatorial testing process (FDA-CIT) to materialize this criterion in practice. At each iteration of FDA-CIT, we detect potential masking effects, heuristically isolate their likely causes (i.e., fault characterization), and then generate new samples that allow previously masked combinations to be tested in configurations that avoid the likely failure causes. The iterations end when the new interaction testing criterion has been satisfied. This paper compares two different fault characterization approaches – an integral part of the proposed approach, and empirically assesses their effectiveness and efficiency in removing masking effects on two widely used open source software systems. It also compares FDA-CIT against error locating arrays, a state of the art approach for detecting and locating failures. Furthermore, the scalability of the proposed approach is evaluated by comparing it with perfect test scenarios, in which all masking effects are known a priori. Our results suggest that masking effects do exist in practice, and that our approach provides a promising and efficient way to work around them, without requiring that masking effects be known a priori.

Index Terms—Combinatorial testing, adaptive testing, covering arrays, software quality assurance

1 Introduction

Software customization, through the modification of run-time or compile-time preferences, allows users to make controlled variations to how their software behaves. Customizable systems such as web servers (e.g., Apache), databases (e.g., MySQL), application servers (e.g., Tomcat) or office applications (e.g., MS Word) which have dozens or even hundreds of options to choose from, can have an enormous number of configurations. For instance, the configuration space of just the optimizer variations for the compiler GCC has 4.6×10^{61} configurations, and the part of the Apache server that controls directory-level accesses has 1.8×10^{55} configurations [14].

While validating the correctness of the system across its entire configuration space is ideal and desirable, exhaustive testing of all configurations is generally infeasible. As the number of configurations grows exponentially with the number of configuration options, the number of possible configurations is typically far beyond the available resources to run the test cases in a timely manner.

One solution approach, called combinatorial interaction testing (CIT), systematically samples the configuration space and tests only the selected configurations [2], [12], [18], [25], [43]. CIT techniques work by first defining a model of the system's configuration space – the set of valid ways that it can be configured. Typically, this model includes a set of configuration options, each of which can take on a small number of option settings, and a set of system-wide inter-option constraints which specify configurations, known a priori, to be invalid. Given this model, CIT methods next compute a small set of valid configurations, a t-way *covering array*, in which each possible combination of option settings for every combination of t options appears at least once [12]. Finally, the system is tested by running its test suite on each configuration in the covering array.

In previous work Yilmaz [42] introduced a new CIT approach, called *test case-aware* CIT, which allows developers to specify not only system-wide constraints, but test case-specific constraints as well. At a high level, a test case-aware covering array can be considered to be a set of traditional

- C. Yilmaz is with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul 34956, Turkey. E-mail: cyilmaz@sabanciuniv.edu.
- E. Dumlu is with Borsa Istanbul, Istanbul, Turkey. She was with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul 34956, Turkey at the time of the work. E-mail: emine.dumlu@borsaistanbul.com.
- M.B. Cohen is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, NE 68558. E-mail: myra@cse.unl.edu.
- A. Porter is with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: aporter@cs.umd.edu.

Manuscript received 15 Oct. 2012; revised 2 July 2013; accepted 19 Oct. 2013; date of publication 4 Nov. 2013; date of current version 24 Feb. 2014. Recommended for acceptance by T. Menzies. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSE.2013.53

Emine Dumlu
M.S., 2011

Cost-aware combinatorial testing

VALID 2012 : The Fourth International Conference on Advances in System Testing and Validation Lifecycle

Cost-Aware Combinatorial Interaction Testing

Gulsen Demiroz and Cemal Yilmaz
Faculty of Engineering and Natural Sciences
Sabanci University, Istanbul 34956, Turkey
Email: {gulsend,cyilmaz}@sabanciuniv.edu

Abstract—The configuration spaces of modern software systems are often too large to test exhaustively. Combinatorial interaction testing approaches, such as covering arrays, systematically sample the configuration space and test only the selected configurations. Traditional t-way covering arrays aim to cover all t-way combinations of option settings in a minimum number of configurations. By doing so, they assume that the testing cost of a configuration is the same for all configurations. In this work, we however argue that, in practice, the actual testing cost may differ from one configuration to another and that accounting for these differences can improve the cost-effectiveness of covering arrays. We first introduce a novel combinatorial object, called a *cost-aware covering array*. A t-way cost-aware covering array is a t-way covering array that minimizes a given cost function. We then provide a framework for defining the cost function. Finally, we present an algorithm to compute cost-aware covering arrays for a simple, yet important scenario, and empirically evaluate the cost-effectiveness of the proposed approach. The results of our empirical studies suggest that cost-aware covering arrays, depending on the configuration space model used, can greatly reduce the actual cost of testing compared to traditional covering arrays.

Keywords–Software quality assurance, combinatorial interaction testing, covering arrays.

I. Introduction

The configuration spaces of configurable software systems are often too large to test exhaustively. The number of possible configurations is often far beyond the available resources to test the entire configuration space in a timely manner, e.g., for regression testing.

Combinatorial interaction testing (CIT) approaches take as input a configuration space model. The model includes a set of configuration options, each of which can take on a number of option settings. Given a configuration space model, the sampling is done by computing a combinatorial object, called a *covering array* (CA). Given a configuration space model, a t-way covering array is a set of configurations, in which each possible combination of option settings for every combination of t options appears at least once [6].

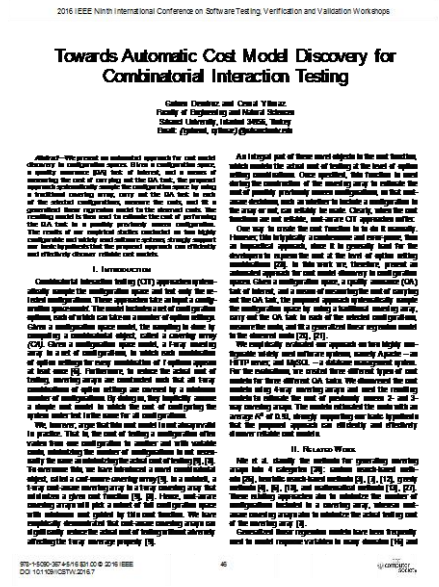
The basic justification for covering arrays is that they can cost-effectively exercise all system behaviors caused by the settings of t or fewer options. The results of many empirical studies strongly suggest that a majority of option-related failures in practice are caused by the interactions among only a small number of configuration options and that traditional t-way covering arrays, where t is much smaller than the number of options, are an effective and efficient way of revealing such failures [2], [6], [9], [10].

Existing approaches construct a t-way covering array in such a way that all valid t-way combinations of option settings are covered by using a minimum number of configurations. By doing so, these approaches implicitly assume a simple cost model where the cost of configuring the system under test is the same for all configurations.

In this work, we argue that this cost model is not always valid in practice. First, we observe that the configuration cost often varies from one configuration to other. For example, in a study conducted on MySQL – a widely-used and highly-configurable database management system, we observed that the cost of configuring the MySQL Community Server (a core component of the system) with its default configuration took about 6 minutes on average (on an 8-core Intel Xeon 2.53GHz CPU with 32 GB of RAM, running CentOS 6.2 operating system). On the other hand, configuring the system with NDB cluster storage support – a feature that enables clustering of in-memory databases, and with embedded server support – a feature that makes it possible to run a full-featured MySQL server inside a client application, took about 9 minutes, as these features needed to be compiled into the system. Therefore, in a covering array, reducing the number of configurations that include these features, without adversely affecting the coverage of option setting combinations, can significantly reduce the amount of time required for testing. However, existing approaches do not take actual testing costs into account when computing covering arrays.

Second, we observe that highly configurable systems often have reusable components, which, once configured, can be used in other configurations with no or very little additional cost. One simple example is the presence of compile-time

 ISBN: 978-1-61208-233-2 9



2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops

Towards Automatic Cost Model Discovery for Combinatorial Interaction Testing

Gulsen Demiroz and Cemal Yilmaz
Faculty of Engineering and Natural Sciences
Sabanci University, Istanbul 34956, Turkey
Email: {gulsend, cyilmaz}@sabanciuniv.edu

Abstract—We present an automated approach for cost model discovery in configuration spaces. Given a configuration space, a quality assurance (QA) task of interest, and a means of measuring the cost of carrying out the QA task, the proposed approach systematically samples the configuration space by using a traditional covering array, carry out the QA task for each of the selected configurations, measure the costs, and fit a generalized linear regression model to the observed costs. The resulting model is then used to estimate the cost of performing the QA task for a possibly previously unseen configuration. The results of our empirical studies conducted on two highly configurable and widely used software systems, strongly support our basic hypothesis that the proposed approach can efficiently and effectively discover reliable cost models.

I. Introduction

Combinatorial interaction testing (CIT) approaches systematically sample the configuration space and test only the selected configurations. These approaches take as input a configuration space model. The model includes a set of configuration options, each of which can take on a number of option settings. Given a configuration space model, the sampling is done by computing a combinatorial object, called a *covering array* (CA). Given a configuration space model, a Fway covering array is a set of configurations, in which each possible combination of option settings for every combination of t options appears at least once [6]. Furthermore, to reduce the actual cost of testing, covering arrays are constructed such that all t-way combinations of option settings are covered by a minimum number of configurations. By doing so, they implicitly assume a simple cost model in which the cost of configuring the system under test is the same for all configurations.

We, however, argue that this cost model is not always valid in practice. That is, the cost of testing a configuration often varies from one configuration to another and with available tools, minimizing the number of configurations is not necessarily the same as minimizing the actual cost of testing [8], [3]. To circumvent this, we have introduced a novel combinatorial object, called a *cost-aware covering array* [3]. In a nutshell, a t-way cost-aware covering array is a t-way covering array that minimizes a given cost function [3], [8]. Hence, cost-aware covering arrays will pick a subset of full configuration space with minimum cost (picked by this cost function. We have empirically demonstrated that cost-aware covering arrays can significantly reduce the actual cost of testing without adversely affecting the t-way coverage property [3].

An integral part of these novel objects is the cost function, which models the actual cost of testing at the level of option setting combinations. Once specified, this function is used during the construction of the covering array to estimate the cost of possibly previously unseen configurations, on that measure decisions such as whether to include a configuration in the array or not, can reliably be made. Clearly, when the cost functions are not reliable, cost-aware CIT approaches suffer.

One way to specify the cost function is to do it manually. However, this is typically a cumbersome and error-prone, thus an impractical approach, since it is generally hard for the developers to have reliable cost models. As a result of option setting combinations [2]. In this work we, therefore, present an automated approach for cost model discovery in configuration spaces. Given a configuration space, a quality assurance (QA) task of interest, and a means of measuring the cost of carrying out the QA task, the proposed approach systematically samples the configuration space by using a traditional covering array, carry out the QA task for each of the selected configurations, measure the costs, and fit a generalized linear regression model to the observed costs [22], [21].

We empirically evaluated our approach on two highly configurable widely used software systems, namely Apache – an HTTP server, and SQLite – a database management system. For the evaluations, we created three different types of cost models for three different QA tasks. We demonstrated that cost models using 4-way covering arrays and used the resulting models to reliably (i.e., not prohibitively expensive 2- and 3-way covering arrays. The models outperformed the ones with an average R^2 of 0.93, strongly supporting our basic hypothesis that the proposed approach can efficiently and effectively discover reliable cost models.

II. Related Work

Nie et al. classify the methods for generating covering arrays into 4 categories [20]: random search-based methods [26], heuristic search-based methods [4], [12], [13], greedy methods [6], [8], [18], and mathematical methods [19], [27]. These existing approaches aim to minimize the number of configurations included in a covering array, whereas cost-aware covering arrays aim to minimize the actual testing cost of the covering array [3].

Generalized linear regression models have been frequently used to model response variables in many domains [16] and

978-1-5090-367-4/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSTW.2016.7 46

Gulsen Demiroz
Ph.D. candidate

Applied Soft Computing 49 (2016) 1125–1144

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Using simulated annealing for computing cost-aware covering arrays

Gulsen Demiroz *, Cemal Yilmaz
Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul 34956, Turkey

ARTICLE INFO

Article history:
Received 30 December 2015
Received in revised form 11 August 2016
Accepted 12 August 2016
Available online 17 August 2016

Keywords:
Software quality assurance
Combinatorial interaction testing
Covering arrays
Cost-aware testing
Simulated annealing

ABSTRACT

The configuration spaces of software systems are often too large to test exhaustively. Combinatorial interaction testing approaches, such as covering arrays, systematically sample the configuration space and test only the selected configurations. In an attempt to reduce the cost of testing, standard t-way covering arrays aim to cover all t-way combinations of option settings in a minimum number of configurations. By doing so, they simply assume that every configuration costs the same. When the cost varies from one configuration to another, however, minimizing the number of configurations is not necessarily the same as minimizing the cost. To overcome this issue, we have recently introduced cost-aware covering arrays. In a nutshell, a t-way cost-aware covering array is a standard t-way covering array that "minimizes" a given cost function modeling the actual cost of testing. In this work we develop a simulated annealing-based approach to compute cost-aware covering arrays, which takes as input a configuration space model enhanced with a cost function and computes a cost-aware covering array by using two alternating neighboring state generation strategies together with a fitness function expressed as a weighted sum of two objectives: covering all required t-way option setting combinations and minimizing the cost function. To the best of our knowledge, the proposed approach is the first approach that computes cost-aware covering arrays for general, non-additive linear cost functions with multiplicative interaction effects. We evaluate the approach both by conducting controlled experiments, in which we systematically vary the input models to study the sensitivity of the approach to various factors and by conducting experiments using real cost functions for real software systems. We also compare cost-aware covering arrays to standard covering arrays constructed by well-known algorithms and study how fast the construction costs are compensated by the cost reductions provided. Our empirical results suggest that the proposed approach is more effective and efficient than the existing approaches.

1. Introduction

The configuration spaces of configurable software systems are often too large to test exhaustively. In general, the number of possible configurations grows exponentially with the number of configuration options. Therefore, testing all configurations in a timely manner is often far beyond the available resources.

Combinatorial interaction testing (CIT) approaches systematically sample the configuration space and test only the selected configurations. These approaches take as input a *configuration space* model. In its simplest form, this model includes a set of configuration options, each of which can take on a small number of settings. Given a configuration space model, the sampling is typically carried out by computing a combinatorial object, called a *covering array*. A t-way covering array is a set of configurations in which each valid combination of option settings for every combination of t options appears at least once [1]. In this definition, t is often referred to as the *coverage strength*.

The basic justification for using CIT is that t-way covering arrays can (under certain assumptions) reveal all failures caused by the interactions of t or fewer options [2]. The results of many empirical studies suggest that the majority of option-related failures in practice are caused by the interactions between only a small number of options; thus, t is small in practice, typically $2 \leq t \leq 6$ [1,3–5]. Therefore, covering arrays are an efficient and effective way of revealing such failures [1,3–5].

In an attempt to reduce the actual cost of testing, standard covering arrays aim to cover all required t-way combinations of option settings using a minimum number of configurations. By doing so, they assume that every configuration costs the same. However, we empirically demonstrated that the cost may vary from one configuration to another and when it does, minimizing the number of configurations is not necessarily the same as minimizing the

* Corresponding author.
E-mail addresses: gulsend@sabanciuniv.edu (G. Demiroz), cyilmaz@sabanciuniv.edu (C. Yilmaz).

http://dx.doi.org/10.1016/j.asoc.2016.08.022
1568-4946/© 2016 Elsevier B.V. All rights reserved.

Funded by Scientific and Technological Research Council of Turkey (113E546)

Cost-Aware Combinatorial Interaction Testing (Doctoral Symposium)

Gulsen Demiroz*
Faculty of Engineering and Natural Sciences
Sabanci University, Istanbul 34956, Turkey
gulsend@sabanciuniv.edu

ABSTRACT

The configuration spaces of software systems are often too large to test exhaustively. Combinatorial interaction testing approaches, such as covering arrays, systematically sample the configuration space and test only the selected configurations. Traditional t-way covering arrays aim to cover all t-way combinations of option settings in a minimum number of configurations. By doing so, they assume that the testing cost of a configuration is the same for all configurations. In my thesis work, we however argue that, in practice, the actual testing cost may differ from one configuration to another and that accounting for these differences can improve the cost-effectiveness of covering arrays. To this end, we introduced a new novel combinatorial object, called a *cost-aware covering array* where a t-way cost-aware covering array is a t-way covering array that minimizes a given cost function. As part of progress, we developed an algorithm for a simple, yet important scenario, and the results of our empirical studies suggest that cost-aware covering arrays can greatly reduce the actual cost of testing compared to traditional covering arrays. We also defined a framework for defining the cost function but then we observed that manually creating those cost models is impractical. Hence our first future goal is to develop an approach for automatically discovering cost models for complex configuration spaces. Our second future goal is then to develop algorithms to generate cost-aware covering arrays for more general cost scenarios. Our focus is currently on meta-heuristic search algorithms such as simulated annealing and genetic algorithms to construct cost-aware covering arrays. Another goal is to expand the cost framework to be test-case aware where not every test case is valid for a configuration, hence the cost of running the test suite is actually different for each configuration.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

*Gulsen Demiroz is advised by Asst. Prof. Cemal Yilmaz.

ISSTA'15, July 13-17, 2015, Baltimore, MD, USA
(c) 2015 ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2784775

Generating Cost-Aware Covering Arrays For Free

Mustafa Kemal Taş, Hanefi Mercan, Gülşen Demiröz, Kamer Kaya and Cemal Yilmaz
{mkemaltas,hanefimercan,gulsend,kaya,cyilmaz}@sabanciuniv.edu

Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul

Abstract. Software systems generally have a large number of configurable options interacting with each other. Such systems are more likely to be prone to errors, crashes, and faulty executions that are usually caused by option interactions. To avoid such errors, testing all possible configurations during the development phase is usually not feasible, since the number of all possible configurations is exponential in the order of number of options. A t-way covering array (CA) is a 2-dimensional combinatorial object that helps to efficiently cover all t-length option interactions of the system under test. Generating a CA with a small number of configurations is important to shorten the testing phase. However, the testing cost (e.g. the testing time) may differ from one configuration to another. Currently, most sequential tools can generate optimum CAs in terms of number of configurations, but they are not cost-aware, i.e., they cannot handle the varying costs of configurations. In this work, we implement a parallel, cost-aware CA-generation tool based on a sequential tool, Jenny, to generate lower-cost CAs faster. Experimental results show that our cost-aware CA construction approach can generate 32% and 21% lower cost CAs on average for t=2 and t=3, respectively, compared to state-of-the-art CA-generation tools. Moreover, the cost-awareness comes for free, i.e., we speed up our algorithm by leveraging parallel computation. The cost models and cost reduction techniques we propose could also be adapted for other existing CA generation tools.

Keywords: Software testing; Testing cost; Combinatorial interaction testing; Covering arrays; Cost-aware testing; Parallel covering array generation

1 Introduction

In software testing, testing all possible configurations of the System Under Test (SUT) is not always feasible. For example, a recent version of the Apache v2.2 web server has 172 configurable options and 1.8×10^{55} unique configurations. Hence, a full coverage of all configurations becomes quite expensive and is not feasible. On the other hand, considering that most of the faults occur due to a small number of option interactions [1], testing all such combinations is usually sufficient to detect the faults. Combinatorial Interaction Testing (CIT) [2, 3] is

Gray-box combinatorial testing

Arsalan Javeed
M.S., 2014
Ph.D. candidate

Combinatorial Interaction Testing of Tangled Configuration Options

Arsalan Javeed and Cemal Yilmaz
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul, Turkey
{ajaveed,cyilmaz}@sabanciuniv.edu

Abstract

Traditional t-way covering arrays have been shown to be highly effective at revealing option-related failures caused by the interactions of t or fewer configuration options. We however, argue that their effectiveness suffers in the presence of complex interactions among configuration options, which from now on will be referred to as tangled options. *To overcome this shortcoming, we propose an approach in which 1) the source code of the system under test is analyzed to figure out how configuration options interact with each other, 2) the analysis results are used to determine the combinations of option settings as well as the conditions under which these combinations must be tested, and 3) a "minimal" interaction test suite covering all the required combinations, is created. To evaluate the proposed approach, we conducted a set of feasibility studies on two highly configurable software systems. The results we have obtained from these studies support our basic hypothesis that traditional covering arrays suffer in the presence of tangled options and that this shortcoming can be overcome by turning CIT from a black-box approach to a gray-box approach.*

1. Introduction

Modern software systems, such as web servers (e.g., Apache), databases (e.g., MySQL), application servers (e.g., Tomcat) or operating systems (e.g., Linux), frequently embody hundreds or even thousands of configuration options that allow end-users to make controlled variations to how their software behaves. While validating the correctness of the system across its entire configuration space is ideal and desirable, exhaustive testing of all configurations is generally infeasible as the number of configurations grows exponentially with the number of configuration options. Therefore, one question that the developers of highly configurable software systems frequently face, is: *Which configurations shall be tested?*

One solution approach is to systematically sample the configuration space by using a combinatorial object, called *a t-way covering array*, and test only the selected configurations. Given a valid configuration space, a *t*-way covering array, where t is often referred to as the *coverage strength*, is a set of configurations, in which each valid combination of option settings for every combination of t options appears at least once [1]. In practice, the coverage strength t is often small, e.g., $2 \leq t \leq 6$, with t=2 is the most common case. Furthermore, for a fixed strength, as the number of options increases, the covering array size represents an increasingly smaller proportion of the whole configuration space. Thus, very large configuration spaces can be efficiently covered.

The basic justification for using covering arrays is that they can effectively reveal all failures caused by the interactions of t or fewer options [7]. However, we argue that covering arrays suffer in the presence of tangled options. That is, when there are complex interactions among configuration options, not all the required combinations of option settings may be tested by traditional *t*-way covering arrays.

Table 1 and 2 illustrate this phenomenon on an example, which is inspired from one of our subject applications used in this work. In this example, we have four compile-time configuration options, namely

. 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
4th International Workshop on Combinatorial Testing (IWCT 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

Unified combinatorial testing

Hanefi Mercan
M.S. 2015
Ph.D. candidate

A Constraint Solving Problem Towards Unified Combinatorial Interaction Testing

Hanefi Mercan and Cemal Yilmaz

Faculty of Engineering and Natural Sciences,
Sabanci University, Istanbul, Turkey
{hanefimercan,cyilmaz}@sabanciuniv.edu

Abstract

Combinatorial Interaction Testing (CIT) approaches aim to reveal failures caused by the interactions of factors, such as input parameters and configuration options. Our ultimate goal in this line of research is to improve the practicality of CIT approaches. To this end, we have been working on developing what we call *Unified Combinatorial Interaction Testing* (U-CIT), which not only represents most (if not all) combinatorial objects that have been developed so far, but also allows testers to develop their own application-specific combinatorial objects for testing. However, realizing U-CIT in practice requires us to solve an interesting constraint solving problem. In this work we informally define the problem and present a greedy algorithm to solve it. Our gaol is not so much to present a solution, but to introduce the problem, the solution of which (we believe) is of great practical importance.

1 Introduction

Software systems frequently embody a wide spectrum of system variabilities that require testing, such as software and hardware configuration options, user inputs, and thread interleavings. However, exhaustively testing all possible variations in a timely manner (if not impossible at all) is generally far beyond the available resources for testing [15]. For this reason, the testing of modern software systems almost always involve sampling enormous variability spaces and testing representative instances of a system's behavior. In practice, this sampling is commonly performed with techniques collectively referred to as combinatorial interaction testing (or CIT) [15, 10].

CIT approaches work by first defining a model of the system's variability space. This model typically includes a set of factors, each of which takes its value from a particular domain, and a (possibly empty) set of inter-option constraints, each of which invalidates certain factor value combinations, as not all possible combinations may be valid in practice. Based on this model, CIT then generates a sample, meeting a specified *coverage criterion*. That is, the sample contains some specified combinations of factors and their values. For instance, a *t-way covering array*, which is a well-known and frequently-used CIT object, requires that each valid combination of factor values for every combination of t factors, appears at least once in the sample [2]. Here, t is often referred to as the coverage strength.

Thank You!