Profiling Deployed Software: Assessing Strategies and Testing Opportunities

Sebastian Elbaum and Madeline Diep

Department of Computer Science and Engineering
University of Nebraska - Lincoln
{elbaum,mhardojo}@cse.unl.edu

October 26, 2004

Abstract

An understanding of how software is employed in the field can yield many opportunities for quality improvements. Profiling released software can provide such an understanding. However, profiling released software is difficult due to the potentially large number of deployed sites that must be profiled, the transparency requirements at a user's site, and the remote data collection and deployment management process. Researchers have recently proposed various approaches to tap into the opportunities offered by profiling deployed systems and overcome those challenges. Initial studies have illustrated the application of these approaches and have shown their feasibility. Still, the proposed approaches, and the tradeoffs between overhead, accuracy, and potential benefits for the testing activity have been barely quantified. This paper aims to overcome those limitations. Our analysis of 1200 user sessions on a 155 KLOC deployed system substantiates the ability of field data to support test suite improvements, assesses the efficiency of profiling techniques for released software, and the effectiveness of testing efforts that leverage profiled field data.

1 Introduction

Software test engineers cannot predict, much less exercise, the overwhelming number of potential scenarios faced by their software. Instead, they allocate their limited resources based on assumptions about how the software will be employed after release. Yet, the lack of connection between in-house activities and how the software is employed in the field can lead to inaccurate assumptions, resulting in decreased software quality and reliability over the system's lifetime. Even if estimations are initially accurate, isolation from what happens in the field leaves engineers unaware of future shifts in user behavior or variations due to new environments until too late.

Approaches integrating in-house activities with field data appear capable of overcoming such limitations. These approaches must profile field data to continually assess and adapt quality assurance activities, considering each deployed software instance as a source of information. The increasing software pervasiveness and connectivity levels[1] of a constantly-growing pool of users coupled with these approaches offers a unique opportunity to gain a better understanding of the software's potential behavior.

Early commercial efforts have attempted to harness this opportunity by including built-in reporting capabilities in deployed applications that are activated in the presence of certain failures (e.g, Software Quality Agent from Netscape [24], Traceback [17], Microsoft Windows Error Reporting API). More recent approaches, however, are designed to leverage the deployed software instances throughout their execution, and also to consider the levels of transparency required when profiling users' sites, the management of instrumentation across the deployed instances, and issues that arise from remote and large scale data collection. For example:

- The Perpetual Testing project produced the residual testing technique to reduce the instrumentation based on previous coverage results [29, 31].
- The EDEM prototype provided a semi-automated way to collect user-interface feedback from remote sites when it does not meet an expected criterion [15].
- The Gamma project introduced an architecture to distribute and manipulate instrumentation across deployed software instances [14].
- The Skoll project presented an architecture and a set of tools to distribute different job configurations across users [34, 33, 22].

Although these efforts present reasonable conjectures, we have barely begun to quantify their potential benefits and costs. Most publications have illustrated the application of isolated approaches [15], introduced supporting infrastructure [34], or explored a technique's feasibility under particular scenarios (e.g., [12, 18]). (Previous empirical studies are summarized in Section 2.5.) Given that feasibility has been shown, we must now quantify the observed tendencies, investigate the tradeoffs, and explore whether the previous findings are valid at a larger scale.

[1] Nielsen reported that 35 million Americans had broadband Internet access in 2003 [25].

This paper's contributions are two fold. First, it presents a family of three empirical studies that quantify the efficiency and effectiveness of profiling strategies for released software. Second, through those studies, it proposes and assesses several techniques that employ field data to drive test suite improvements and it identifies factors that can impact their potential cost and benefits.

In the following section, we abstract the essential attributes of existing profiling techniques for released software to organize them in strategies and summarize the results of previous empirical studies. Section 3 describes the research questions, object preparation, design and implementation, metrics, and potential threats to validity. Section 4 presents results and analysis. Section 5 provides additional discussion and conclusions.

2 Profiling Strategies for Released Software

Researchers have enhanced the efficiency of profiling techniques through several mechanisms: (1) performing up-front analysis to optimize the amount of instrumentation required [3], (2) sacrificing accuracy by monitoring entities of higher granularity or by sampling program behavior [10, 11], (3) encoding the information to minimize memory and storage requirements [30], and (4) repeatedly targeting the entities that need to be profiled [1]. The enumerated mechanisms gain efficiency by reducing the amount of instrumentation inserted in a program through the analysis of the properties it exhibited in a controlled in-house environment.

Profiling techniques for released software [14, 15, 29, 34], however, must consider the efficiency challenges and the new opportunities introduced by a potentially large pool of users. Profiling deployed software is faced with the challenge of operating under the threshold of being noticeable to the user. Collecting profile data from thousands of instances is faced with the challenge of handling and analyzing overwhelming amounts of data. The ultimate challenge is to create profiling strategies that overcome those challenges and, in our case, effectively leverage the field data to improve the testing activities.

This paper investigates three of such profiling strategies. The strategies abstract the essential elements of existing techniques helping to provide an integrated background, and also facilitating formalization, analysis of tradeoffs, and comparison of existing techniques and their implementation. The next section describes the *full* strategy which constitutes our baseline technique, the following three sections describe the profiling strategies for released software, and Section 2.5 summarizes the related applications and empirical studies.

2.1 Full Profiling

Given a program P and a class of events to monitor C, this approach generates P' by incorporating instrumentation code into P to enable the capture of ALL events in C and a transfer mechanism T to transmit the collected data to the organization at the end of each execution cycle (e.g., after each operation, when a fatal error occurs, or when the user terminates a session).

Capturing all the events at all user sites demands extensive instrumentation, increasing program size and execution overhead (reported to range between 10% to 390% [2, 21]). However, this approach also provides the maximum amount of data for a given set C serving as a baseline for the analysis of the other three techniques specific to release software. We investigate the raw potential of field data through the *full* technique in Section 4.1.

2.2 Targeted Profiling

Before release, software engineers develop an understanding about the program behavior. Factors such as software complexity and schedule pressure limit the levels of understanding they can achieve. This situation leads to different levels of certainty about the behavior of program components. For example, when testers validate a program with multiple configurations, they may be able to gain certainty about a subset of the most used configurations. Some configurations, however, might not be (fully) validated.

To increase profiling transparency in deployed instances, software engineers may target components for which the behavior is not sufficiently understood. Following with our example about a software with multiple configurations, engineers aware of the overhead associated with profiling deployed software could aim to profile just those least understood or most risky configurations.

More formally, given program P, a class of events to profile C, a list of events observed (and sufficiently understood) in-house $C_{observed}$ where $C_{observed} \subset C$, this technique generates a program P' with additional instrumentation to profile all events in $C_{targeted} = C - C_{observed}$. Observe that $|C_{targeted}|$ determines the efficiency of this technique by reducing the necessary instrumentation, but also bounding what can be learned from the field instances. In practice, the gains generated through targeted profiling are intrinsically tied to the engineers ability to determine what is worth targeting. Note also that as the certainty about the behavior of the system or its components diminishes, $|C_{observed}| \rightarrow 0$, this strategy's performance approximates *full*.

As defined, this strategy includes the residual testing technique [29] where $C_{observed}$ corresponds to statements observed in-house, and it includes the distribution scheme proposed by Skoll [22, 34, 33] where $C_{targeted}$ corresponds to target software configurations that require field testing.

2.3 Profiling with Sampling

Statistical sampling is the process of selecting a suitable part of a population for determining the characteristics of the whole population. Profiling techniques have adapted the sampling concept to reduce execution costs by repeatedly sampling across space and time. Sampling across space consists of profiling a subset of the events following a certain criterion (e.g., hot paths). Sampling across time consists of obtaining a sample from the population of events at certain time intervals [4]. Common profiling utilities like *gprof* follow this approach, stopping execution at fixed intervals to determine where the cycles are being spent [11].

When considering released software, we find an additional sampling dimension: the instances of the program running in the field. We could, for example, sample across the population of deployed instances, profiling the behavior of a group of users. This is advantageous because it would only add the profiling overhead to a subset of the population. Still, the overhead on this subset of instances could substantially affect user's activities, biasing the collected information.

An alternative sampling mechanism could consider multiple dimensions. For example, we could stratify the population of events to be profiled following a given criterion (e.g., events from the same functionality) and then sample across the subgroups of events, generating a version of the program with enough instrumentation to capture just those sampled events. Then, by repeating the sampling process, different versions of P' can be generated for distribution. Potentially, each user could obtain a slightly different version aimed to capture a particular event sample. [2]

More formally, given program P and a class of events to monitor C, the stratified sampling strategy performs the following steps: 1) it identifies s strata of events $C_1, C_2, ..., C_i, ..., C_s$, 2) it selects a total of N events from C by randomly picking n_i from C_i, where n_i is proportional to the stratum size, and 3) it generates P' to capture the selected events. As N gets smaller, P' contains less instrumentation, enhancing transparency but possibly sacrificing accuracy. By repeating the sampling and generation process, $P'', P''', ..., P^m$ are generated, resulting in versions with various suitable instrumentation patterns available for deployment.[3]

There is an important tradeoff between the event sample size N and the number of deployed instances. Maintaining N constant while the number of instances increases results in constant

[2] An alternative procedure could first stratify the user population, and then sample across events. However, we conjecture that finding strata of known events is easier than finding subgroups of an unknown or shifting user population.

[3] In this work we focused on two particular dimensions: space and deployed instances. However, note that sampling techniques on time could be applied on the released instances utilizing the same mechanism.

profiling transparency across sites. As the number of deployed instances increases, however, the level of overlap in the collected data across deployed sites is also likely to increase. Section 4.2 investigates how to leverage this overlap to reduce N, gaining transparency at each deployed site by collecting less data while compensating by profiling more deployed instances.

As defined, our sampling strategy provides a statistical framework for various algorithms implemented by the Gamma research effort [18, 28]. One distinct advantage of this framework, however, is that it generalizes the procedure to sample across deployed sites which, as we shall see in Section 4.2, offers further opportunities to improve efficiency.

2.4 Trigger Data Transfers on Anomalies

Transferring data from deployed sites to the organization can be costly for both parties. For the user, data transfer implies at least additional computation cycles to marshal and package data, bandwidth to actually perform the transfer, and a likely decrease in transparency. For an organization with thousands of deployed software instances, data collection might become a bottleneck. Even if this obstacle could be overcome with additional collection devices (e.g., a cluster of collection servers), processing and maintaining such a data set could prove expensive. Triggering data transfers in the presence of anomalies can help to reduce these costs.

Employing anomaly detection implies the existence of a baseline behavior considered nominal or normal. When targeting released software, the nominal behavior is defined by what the engineers know or understand. For example, engineers could define an operational profile based on the execution probability exhibited by a set of beta testers [23]. A copy of this operational profile could be embedded into the released product so that deviations from its values trigger a data transfer. Sessions that fit within the operational profile would only send a confirmation to the organization, increasing the confidence in the estimated profile; sessions that fall outside an specified operational profile range are completely transferred to the organization for further analysis (e.g., determine whether the anomaly indicates a potential problem, update the profile if the anomaly was due to an incomplete in-house assessment).

We now define the approach more formally. Given program P, a class of events to profile C, an in-house characterization of those events C_{house}, a tolerance to deviations from the in-house characterization $C_{houseTolerance}$, this technique generates a program P' with additional instrumentation to monitor events in C, and a detection algorithm to identify when field behavior C_{field} deviates from $[C_{house} \pm C_{houseTolerance}]$. When such deviation is detected, session data is transferred to the organization. Note that this definition of trigger by anomaly includes the type of behavioral "mismatch" trigger mechanism used by EDEM [15] by making $C_{houseTolerance} = 0$.

There are many interesting tradeoffs in defining and detecting deviations from C_{house}. For example, there is a tradeoff between the level of investment on the in-house software characterization and the number of false negatives reported from the field. Also, there are myriad of algorithms to detect anomalies, trading detection sensitivity and effectiveness with execution overhead. We investigate some of these tradeoffs in Section 4.3.

2.5 Applications and Previous Empirical Studies

The efforts to develop profiling techniques for released software have aimed for different goals and have been supported by different mechanisms. Hilbert and Redmiles [15, 16] utilized walk-through scenarios to demonstrate the concept of internet mediated feedback. The scenarios illustrated how software engineers could improve user interfaces through the collected information. The scenarios reflected the authors' experiences in a real context, but they did not constitute empirical studies.

Pavlopoulou and Young empirically evaluated the efficiency gains of the residual testing technique on programs of up to 4KLOC [29]. The approach considered instrumentation probe removal, where a probe is the snippet of code incorporated into P to profile a single event. Executed probes were removed after each test was executed, showing that instrumentation overhead to capture coverage can be greatly reduced under certain conditions (e.g., similar coverage patterns across test cases, non-linear program structure). Although incorporating field data into this process was discussed, this aspect was not empirically evaluated.

The Gamma group performed at least two empirical studies to validate the efficiency of their distributed instrumentation mechanisms. Bowring et al. [18] studied the variation in the number of instrumentation probes and interactions, and the coverage accuracy for two deployment scenarios. For these scenarios, they employed a 6KLOC program and simulated users with synthetic profiles. A second study by the same group of researchers lead by Orso [27] employed the created infrastructure to deploy a 60KLOC system and gathered profile information on 11 users (7 from the research team) to collect 1100 sessions. The field data was then used for impact analysis and regression testing improvement. The findings indicate that field data can provide smaller impact sets than slicing and truly reflect the system utilization, while sacrificing precision. The study also highlighted the potential lack of accuracy of in-house estimates, which can also lead to a more costly regression testing process.

The Skoll group has also started to perform studies to study the feasibility of their infrastructure to utilize user participation in improving quality assurance process on two large open source projects (ACE and TAO) [34]. The feasibility studies reported on the infrastructure's capability in detecting failures in several configuration settings. Yilmaz et al. have also proposed an approach using covering arrays to steer the sampling of the configuration settings to be tested [33]. Early assessments performed through simulations across a set of approximately ten workstations revealed a reduction in the number of configurations to be tested while identifying some faulty configurations not found by the original developers [22].

Overall, when revisiting the previous studies in terms of the profiling strategies we find that: 1) the transfer on anomaly strategy has not been validated through empirical studies, 2) the targeted profiling strategy has not been validated with deployment data and its efficiency analysis included just four small programs, 3) the strategy involving sampling has been validated more extensively in terms of efficiency but the effectiveness measures were limited to coverage, and 4) each assessment was performed in isolation. Our studies address those weaknesses by improving on the following items:

- Target object and subjects. Our empirical studies are performed on a 155KLOC program utilized by 30 users, providing the most comprehensive empirical setting yet to study this topic.
- Comparison and integration of techniques within the same scenarios. We analyze the benefits and costs of field data obtained with full instrumentation and compared it against techniques utilizing targeting profiling, sampling profiling, a combination of targeting and sampling, and anomaly driven transfers.
- Assessment of effectiveness. We assess the potential of field data in terms of coverage gains, additional fault detection capabilities, potential for invariant refinements, and overall data capture.
- Assessment of efficiency. We count the number of instrumentation probes required and executed, and also measure the number of required data transfers (a problem highlighted but not quantified in [15]).

3 Empirical Study

This section introduces the research questions that serve to scope this investigation. The metrics, object of study, and the design and implementation follow. Last, we identify the threats to validity.

3.1 Research Questions

We are interested in the following research questions.

RQ1: What is the potential benefit of profiling deployed software instances? In particular, we investigate the coverage, fault detection effectiveness, and invariant refinements gained through the generation of a test suite based on field data.

RQ2: How effective and efficient are profiling techniques designed to reduce overhead at each deployed site? We investigate the tradeoffs between efficiency gains (as measured by the number of probes required to profile the target software and the probes executed in the field), coverage gains, and data loss.

RQ3: Can anomaly based triggers reduce the number of data transfers? What is the impact on the potential gains? We investigate the effects of triggering transfers when a departure from an operational profile is detected, when an invariant is violated, and when a stochastic model predicts that the sequence of events is likely to lead to a failure state.

3.2 Object

We selected the popular[4] program *Pine* (Program for Internet News and Email) as the object of the experiment. *Pine* is one of the numerous programs to perform mail management tasks. It has several advanced features such as support for automatic incorporation of signatures, internet newsgroups, transparent access to remote folders, message filters, secure authentication through SSL, and multiple roles per user. In addition, it supports tens of platforms and offers flexibility for a user to personalize the program by customizing configuration files. Several versions of *Pine* source code are publicly available. For our study we primarily use the Unix build, version 4.03, which contains 1373 functions and over 155 thousand lines of code including comments.

[4] *Pine* had 23 million users worldwide as of March 2003 [26].

3.2.1 Test Suite

To evaluate the potential of field data to improve the in-house testing activity we required an initial test suite on which improvements could be made. Since *Pine* does not come with a test suite, two graduate students, who were not involved in the current study, developed a suite by deriving requirements from *Pine*'s man pages and user's manual, and then generating a set of test cases that exercised the program's functionality. Each test case was composed of three sections: 1) a setup section to set folders and configurations files, 2) a set of Expect [20] commands that allows to test interactive functionality, and 3) a cleanup section to remove all the test specific settings. The test suite consisted of 288 automated test cases, containing an average of 34 Expect commands. The test suite took an average of 101 minutes to execute on a PC with an Athlon 1.3 processor, 512MB of memory, running Redhat version 7.2. When function and block level instrumentation was inserted, the test suite execution required 103 and 117 minutes respectively (2% and 14% overhead). Overall, the test suite covered 835 functions.

3.2.2 Faults

To quantify potential gains in fault detection effectiveness we required the existence of faults. We leveraged a parallel research effort by our team [7] that had resulted in 43 seeded faults in four posterior versions of *Pine*: 4.04, 4.05, 4.10, and 4.20. (The seeding procedure follows the one detailed in [8].) We then utilized these faults to quantify the fault detection effectiveness of the test suites that leveraged field data.

3.2.3 Invariants

To quantify potential gains we also considered the dynamic invariant refinement that can be achieved with the support of field data. Since invariants specify the program operational behavior, we conjectured that novel field behavior could be employed to trigger data transfers and refine invariants generated in-house. Because *Pine* did not have built-in invariants, we used our in-house test suite to dynamically discover its invariants following a similar procedure to that utilized by Ernst et al. [9] with the assistance of of Daikon [13]. The resulting invariants were located at the entry and exit point of methods and classes called program points (PP). After removing "zero-sample" program points (program points that are never exercised by the input) and redundant invariants we obtained 2113 program points and 56212 invariants.

3.3 Study Design and Implementation

The overall empirical approach was driven by the research questions and constrained by the costs of collecting data for multiple deployed instances. Throughout the study design and implementation process, we strived to achieve a balance between the reliability and representativeness of the data collected from deployed instances under a relatively controlled environment, and the costs associated with obtaining such a data set. As we shall see, combining a controlled deployment and collection process with aposteriori simulation of different scenarios and techniques helped to reach such a balance (potential limitations of this approach are presented under threats to validity in Section 3.4).

We performed the study in three major phases: (1) object preparation, (2) deployment and data collection, and (3) processing and simulation.

The first phase consisted of instrumenting *Pine* to enable a broad range of data collection. The instrumentation is meant to capture functional coverage information [5], operational traces, accesses to environmental variables, and changes in the configuration file occurring in a single session (a session is initiated when the program starts and finishes when the user exits). In addition, to enable further validation activities (e.g., test generation based on user's session data), the instrumentation also enables the capture of various session attributes associated with user operations (e.g., folders, number of emails in folders, number and type of attachments, errors reported on input fields). At the end of each session, the collected data is time-stamped, marshaled, and transferred to the central repository. For anonimization purposes, the collected session data is packaged and labeled with the encrypted sender's name at the deployed site and the process of receiving sessions is conducted automatically at the server to reduce the likelihood of associating a data package with its sender.

We conducted the second phase of the study in two steps. First, we deployed the instrumented version of *Pine* at five "friendly" sites. For two weeks we used this preliminary deployment to verify the correctness of the installation scripts, data capture process and content, magnitude and frequency of data transfer, and the transparency of the de-installation process. After this initial refinement period, we proceeded to expand the sample of users. The target population corresponded to the approximately 60 students in our Department's largest research lab. After promoting the study for a period of two weeks, 30 subjects volunteered to participate in the study (members from

[5] We made a decision to capture functional level data in the field because of its relative low overhead (see Section 3.2).

our group were not allowed to participate). The study's goal, setting, and duration (45 days) was explained to each one of the subjects, and the same fully-instrumented version of the *Pine*'s package was made available for them to install. At the termination date, 1193 user sessions had been collected, an average of 1 session per user per day (no sessions were received during 6 days due to data collection problems).

The last phase consisted of employing the collected data to support different studies and simulating different scenarios that could help us answer the research questions. The particular simulation details such as the manipulated variables, the nuisance variables and the assumptions, vary depending on the research question, so we address them individually within each study.

3.4 Threats to Validity

This study, like any other, has some limitations that could have influenced the results. Some of these limitations are unavoidable consequences of the decision to combine an observational study with simulation. However, given the cost of collecting field data and the fundamental exploratory questions we are pursuing, these two approaches offered us a good balance between data representativeness and power to manipulate some of the independent variables.

By collecting data from 30 deployed instances of *Pine* during a period of 45 days, we believe to have performed the most comprehensive study of this type. Our program of study is representative of many programs in the market, limiting threats to external validity. In spite of the gaps in the data collection process due to data corruption and server "down-time", we believe the large number of sessions collected overshadows the data loss and does not constitute a significant threat.

Regarding the selected pool of users, the subjects are students in our Department, but we did not exercise any control during the period of study, which gives us confidence that they behaved as any other user would under similar circumstances. Still, more studies with other programs and subjects are necessary to confirm the results we have obtained. For example, we must include subjects that exercise the configurable features of *Pine* and we need to distribute versions of the program for various configurations.

Our simplifying assumptions about the deployment management is another threat to external validity. Although the assumptions are reasonable and could be restated for further simulation if needed, empirical studies specifically including various deployment strategies are required [32]. Furthermore, our studies assume that the incorporation of instrumentation probes and anomaly based triggers do not result in additional program faults and that repeated deployments are technically and economically feasible in practice.

We are also aware of the potential impact of observational studies on a subject's behavior. Although we clearly stated our goals and procedures, subjects could have been afraid to send certain type of messages through our version of *Pine*. This was a risk we were willing to take to increase the chances of exploring different scenarios through simulation. Overall, gaining users trust and willingness to be profiled is a key issue for the proposed approaches to succeed and should be the focus of future studies.

The in-house validation process helped to set a baseline for the assessment of the potential of field data (RQ1) and the anomaly based triggers for data transfers (RQ3). As such, the quality of the in-house validation process is a threat to internal validity which could have affected our assessments. We partially studied this factor by carving weaker test suites from an existing suite (Section 4.1) and by considering different number of users to characterize the initial operational profiles (Section 4.3). Still, the scope of our findings are affected by the quality of the initial suite.

When implementing the techniques we had to make certain choices. Some of these choices may limit the generality of our findings. For example, we implemented six test case generation techniques that utilized different source data or follow different procedures, we employed the invariant discovery tool with one particular configuration setting, and we choose one particular type of clustering algorithm. Different sources of data, settings, or algorithms could have resulted in a different outcome. To limit this threat, we considered the cost-benefits and potential applicability resulting from our choices.

Last, the metrics we have selected are just a subset of the potential metrics that could have been used to assess the potential of field data and the performance of profiling strategies. For example, our study was not designed to capture bandwidth consumption, throughput reduction due to overhead, or deployment costs. Instead, at this empirical stage, our assessment consisted of quality measures (e.g., fault detection, coverage gain, invariant refinement, data loss) and their tradeoffs with performance surrogate measures such as the number of probes required, number of of probes executed, or the number of transfers required.

4 Studies' Setting and Results

The following sections address our research questions. Each section contains two parts, one explaining the simulation setting and one presenting the findings.

4.1 RQ1 Study: Field Data Driven Improvement

This first study addresses RQ1, aiming to provide a better understanding of the potential of field data to improve a test suite. This study assesses three different mechanisms to harness the potential of field data for test suite improvement as measured by coverage, fault detection, and invariants detection gains.

4.1.1 Simulation Setting

For this study, we assume a *full* profiling strategy was employed to collect coverage data on each deployed instance. We also assume that the potential benefits are assessed at the end of the collection process without loss of generality (the same process could be conducted more often).

We consider three groups of test case generation procedures. These groups appeared in Table 1. First, we defined a procedure that can generate test cases to reach all the entities executed by the users, including the ones missed by the in-house test suite. We call this hypothetical procedure "Potential" because it sets an upper bound on the performance of test suites generated based on field data.

Second, we consider four automated test case generation procedures that translate each user session into a test case. The procedures vary in the data they employ to recreate the conditions at the user site. For example, to generate a test case with the $A3$ procedure the following field data is employed: 1) a sequence of commands replaying the ones employed by the user and dummy values to complete necessary fields (e.g., email's destination, subject, content), 2) the user initial *Pine* configuration, and 3) an initial setup where the configuration file and mailbox content is matched as closely as possible to the one in the session. Per definition, all the automated test case generation procedures result in 1193 test cases (one test case per user session).

Third, we allow a tester to enhance the two best automatic test case generation procedures ($A3$ and $A4$) through the analysis of specific trace data. We constrained the tester to traces from field sessions that provided coverage gains not captured by the automated procedures. This enhancement resulted in 572 additional test cases for $M1$ and 315 additional test cases for $M2$.

Note that both, the automated generated tests and the ones generated with the tester's assistance, are not meant to exactly reproduce the behavior observed in the field; these procedures simply attempt to leverage field data to enhance an existing test suite with cases that approximate field behavior to certain degree.

Table 1: Test Case Generation Procedures Based on Field Data

Procedure	**Description**
Potential	Best potential approximation of user activity and context.
Automatic translation 1 ($A1$)	Menu sequence + Generic Setup
Automatic translation 2 ($A2$)	Menu sequence + Pine's Configuration File + Generic Setup
Automatic translation 3 ($A3$)	Menu sequence + Pine's Configuration File + Custom Setup
Automatic translation 4 ($A4$)	Menu sequence + Pine's Configuration File + Environment Configuration + Custom Setup
$A3$ + Manual enhancement 1 ($M1$)	Menu sequence + Pine's Configuration File + Custom Setup + Function Trace
$A4$ + Manual enhancement 2 ($M2$)	Menu sequence + Pine's Configuration File + Environment Configuration + Custom Setup + Function Trace

4.1.2 Results

Coverage. Figure 1 shows how individual sessions contribute coverage. During the 1193 collected user sessions, 128 functions (9.3% of the total) that were not exercised by the original test suite were executed in the field. Intuitively, as more sessions are gathered we would expect it would become harder to cover new functions. This is corroborated by the growth of the cumulative coverage gain line. Initially, field behavior exposes many originally uncovered functions. This benefit, however, diminishes over time suggesting perhaps that field data can benefit testing the most at the early stages after deployment.

Table 2 presents the gains for the three groups of test generation procedures. The first row presents the potential gain of field data if it is fully leveraged. Row $2-5$ present the fully automated translation approach to exploit field data, which can generate a test suite that provides at most 3.5%

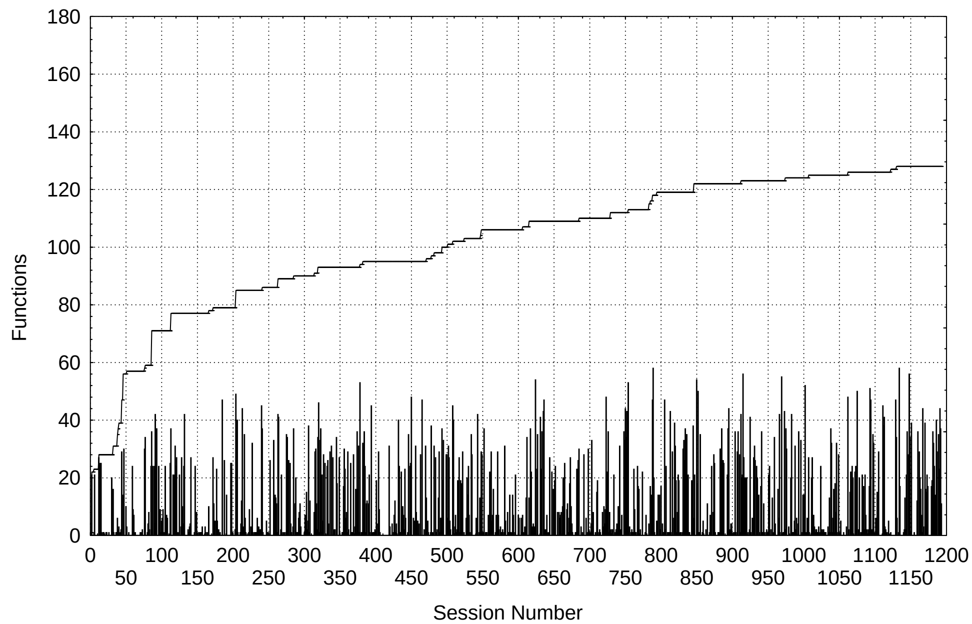

Figure 1: Coverage gain per session and cumulative

function coverage gain. The more elaborated test case generation procedures that require tester's participation (row 6 and 7 in Table 2), provide at most 6.9% gains in terms of additional functions covered.

Although we did not capture block coverage at the deployed instances (na cell in Table 2), we used the test cases generated to measure the block coverage gain. The coverage gains at the block level exercised from 4.8% to 14.8% of additional blocks with the utilization of field data, providing additional evidence about the value of data from deployed instances.

Table 2: Coverage Data

Procedure	**Coverage**	
	Function	**Block**
Potential	128 (9.3%)	na
$A1$	26 (1.9%)	513 (4.8%)
$A2$	31 (2.2%)	697 (6.5%)
$A3$	44 (3.2%)	1068 (10.0%)
$A4$	49 (3.5%)	1153 (10.8%)
$M1$	81 (5.9%)	1328 (12.5%)
$M2$	95 (6.9%)	1328 (14.8%)

Still, as the in-house suite becomes more powerful and just the extremely uncommon execution patterns remained uncovered, we expect that the realization of potential gains will become harder because a more accurate reproduction of the user session may be required. For example, a user session might cover a new entity when special characters are used in an email address. Reproducing that class of scenario would require capturing the content of the email, which we refrained from doing for privacy and performance concerns. This type of situation may limit the potential gains that can be actually exploited.

Fault Detection. The test suites developed utilizing field data also generated gains over the in-house test suite in terms of fault detection capabilities. Table 3 reports the faults seeded in each version of $Pine$, the faults detected through the in-house test suite, and the additional faults found through the test cases generated based on field data. The test suite generated through the automatic procedures discovered up to 7% of additional faults over the in-house test suite, and the procedures requiring the tester's participation discovered up to 26% of additional faults. Field data raised the overall testing effectiveness on future versions (regression testing), as measured by fault detection, from 65% to 86%.

Table 3: Fault detection effectiveness

$Pine$ Version	Fault Seeded	Detected In-house	Additional faults detected					
			$A1$	$A2$	$A3$	$A4$	$M1$	$M2$
v4.04	10	5 (50%)	0 (0%)	0 (0%)	0 (0%)	1 (10%)	2 (20%)	2 (20%)
v4.05	3	3 (100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (33%)
v4.10	15	11 (73%)	0 (0%)	0 (0%)	0 (0%)	1 (7%)	1 (7%)	2 (13%)
v4.20	15	9 (60%)	0 (0%)	1 (7%)	1 (7%)	1 (7%)	2 (13%)	4 (26%)
	43	28 (65%)	0 (0%)	1 (7%)	1 (2%)	3 (7%)	5 (12%)	9 (21%)

Invariant Refinement. Table 4 describes the potential of field data to drive invariant refinements. We compared the invariants resulting from the in-house test suite versus the invariants resulting from the union of the in-house test suite and the test suites generated with the procedures defined in Table 1. The comparison is performed in terms of the number of invariants and the number of program points removed and added. To determine whether the refinements were due to just additional coverage, we also discriminated between invariants and program points added in the functions covered in-house, and the ones in newly executed functions. Invariants can also be generalized to always hold true for every possible exit point of a method in a program instead of only to hold true for a specific exit point in a method. We denote this as generalized PP and specific PP respectively, and use it to further discriminate the refinement obtained with field data .

Table 4: Invariants refinement

Procedure	Criteria	Function Covered	Function Not Covered
$In-house+A1$	Removed Inv.	2275 (4.04%)	na
	Added Inv.	8563 (15.23%)	1531 (2.72%)
	Added Generalized PP	74 (3.50%)	98 (4.63%)
	Removed Specific PP	127 (6.01%)	na
$In-house+A2$	Removed Inv.	2964 (5.27%)	na
	Added Inv.	10134 (18.08%)	1901 (3.38%)
	Added Generalized PP	137 (6.48%)	124 (5.86%)
	Removed Specific PP	154 (7.28%)	na
$In-house+A3$	Removed Inv.	3813 (6.78%)	na
	Added Inv.	20341 (36.18%)	2474 (4.40%)
	Added Generalized PP	184 (8.70%)	137 (6.48%)
	Removed Specific PP	179 (8.47%)	na
$In-house+A4$	Removed Inv.	4612 (8.20%)	na
	Added Inv.	28647 (50.96 %)	2758 (4.91%)
	Added Generalized PP	216 (10.22%)	149 (7.05%)
	Removed Specific PP	194 (9.18%)	na
$In-house+M1$	Removed Inv.	5449 (9.69 %)	na
	Added Inv.	32912 (58.54%)	2941 (5.23%)
	Added Generalized PP	227 (10.74%)	164 (7.76%)
	Removed Specific PP	214 (10.12%)	na
$In-house+M2$	Removed Inv.	6351 (11.29%)	na
	Added Inv.	34108 (60.67%)	3164 (5.62%)
	Added Generalized PP	241 (11.41%)	194 (9.18%)
	Removed Specific PP	226 (10.69%)	na

Table 4 reveals that all the test case generation procedures based on field data help to refine the existing invariants in the programs. The procedures more effective in terms of coverage and fault detection were also the most effective in providing invariant refinements. For example, with the assistance of $M2$ we were able to discard 11% of the 56212 invariants detected with the in-house test suite, add 66% new invariants (6% coming from newly covered functions), and add 21% generalized program points. Even with the simplest automated procedure $A1$, 4% of the invariants were removed, and 18% were added. Most differences between consecutive techniques

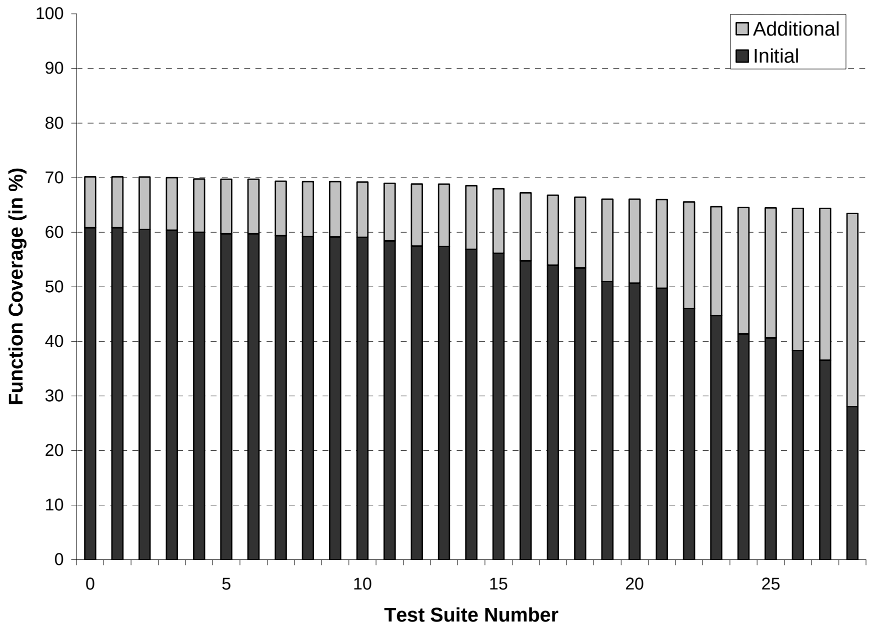

Figure 2: Initial and gained coverage per test suite

(e.g., $A1$ vs. $A2$) were less than 12%, with the exception of the "added invariants" criteria for which increases were quadratic in some cases. For example, when utilizing $A4$ instead of $A3$ there is an increase of 14% in the number of invariants added in the covered functions. This indicates that a closer representation of the user setting is likely not only to result in refined invariants, but also to drastically increase the characterization breath of the system behavior.

Impact of initial test suite coverage. Intuitively, initial test suite coverage would affect the potential effectiveness of the field data to facilitate any improvement. For example, a weaker initial test suite is likely to benefit sooner from field data. To confirm that intuition, we carved several test suites from the initial suite with varying levels of coverage. Given the initial test suite T_0 of size n_0, we randomly choose n_1 test cases from T_0 to generate T_1, where $n_1 = n_0 * potencyRatio$ and $potencyRatio$ values range from 0 to 1. We arbitrarily picked a fixed $potencyRatio$ of 0.96 (equivalent to decreasing T_i by 10 test cases to create T_{i+1}), repeating this process using test suite T_i to generate T_{i+1}, where $T_i \supset T_{i+1}$. This process resulted in 29 test suites with decreasing coverage levels.

We then consider the impact of test suite coverage on potential gains. Figure 2 shows how T_0 initial coverage is 60%, potential gain is 9.3%, and overall coverage of 69%. On the other hand,

T_{28} provides an initial coverage of 28%, potential gains of 35%, resulting in an overall coverage of 63%. Suites with lower initial coverage are likely to benefit more from field data. Note, however, that the overall coverage (initial plus gain) achieved by a weaker test suite is consistently inferior to that of a more powerful test suite. This indicates that, even if the full potential of field data is exploited to generate test cases, more sessions with specific (and perhaps unlikely) coverage patterns may be required to match the results from a stronger in-house suite.

4.2 RQ2 Study: Targeted and Sampled Profile

This study addresses RQ2, providing insights on the reduction of profiling overhead achieved by instrumenting a subset of the entities or sampling across the space of events and deployed instances.

4.2.1 Simulation Setting

We investigate the performance of two profiling techniques: targeted profiling (tp) and sampling profiling (sp). The results from the *full* profiling technique serve as the baseline.

For tp, we assume the software was deployed with enough instrumentation to profile all functions that were not executed during the testing process (considering our most powerful test suite T_0). Since we stored the coverage information for each session, we can simulate the conditions in which only a subset of those elements are instrumented. We also assume the same instrumented version is deployed at all sites during the duration of the study.

For sp, we have to consider two parameters: the number of strata defined for the event population and the number of versions to deploy. For our study, we defined as many strata as program components (each of the 27 files constitutes a component). Regarding the versions for P, we generate multiple versions by incorporating different sets of instrumentation probes in each version. If we assume each user gets a different instrumented version of the program, then the maximum number of versions is 30 (one version per user). The minimum number of versions to apply the sampling across users strategy is two. We also simulated having 5, 10, 15, 20, and 25 deployed versions to observe the trends among those extremes. We denote these sp variations as $sp2$, $sp5$, $sp10$, $sp15$, $sp20$, $sp25$, and $sp30$. To get a better understanding of the variation due to sampling, we performed this process on each sp variation five times, and then performed the simulation. For example, for $sp2$, we generated five pairs of two versions, where each version in a pair had half of the functions instrumented, and the selection of functions was performed by randomly sampling functions across the identified components.

The simulation results for these techniques were evaluated through two measures. The number of instrumentation probes required and the ones executed in the field constitute our efficiency measures. The potential coverage gains generated by each technique constitute our effectiveness measure.

4.2.2 Results

Efficiency: number of probes required. The scatterplot in Figure 3 depicts the number of instrumentation probes and the potential coverage gain for *full*, tp, and a family of sp techniques. There is one observation for the *full* and tp techniques, and the five observations for the sp (corresponding to the five samples that were simulated) have been averaged.

The *full* technique needs almost 1400 probes to provide 9.3% gain. By utilizing the tp technique, and given the in-house test suite we had developed, we were able to reduce the number of probes to 32% or 475 probes from the *full* technique without loss in potential gain, confirming previous conjectures about its potential. The sp techniques provided efficiency gains with a wide range of effectiveness. As more instrumented versions are deployed, the profiling efficiency as measured by the number of instrumentation probes increases. With the most aggressive implementation of the technique, $sp30$, the number of probes per deployed instance is reduced to 3% of the *full* technique. By the same token, a more extensive distribution of the instrumentation probes across a population of deployed sites diminished the potential coverage gains to approximately a third (when each user gets a uniquely instrumented version the coverage gains is below 3%) compared with the *full* technique.

We then proceeded to combine tp and sp to create a hybrid technique, hyb, that performs sampling only on functions that were not covered by the initial test suite. Applying tp on $sp2$, $sp5$, $sp10$, $sp15$, $sp20$, $sp25$, and $sp30$ yielded $hyb2$, $hyb5$, $hyb10$, $hyb15$, $hyb20$, $hyb25$, and $hyb30$, respectively. Hybrid techniques (depicted in Figure 3 as triangles) were as effective as sp techniques in terms of coverage gains but even more efficient. For example, $hyb2$ cut the number of probes by approximately 70% compared to $sp2$. However, when a larger number of instrumented versions are deployed, the differences between the hyb and the sp techniques become less obvious as the chances of obtaining gains in an individual deployed site diminishes.

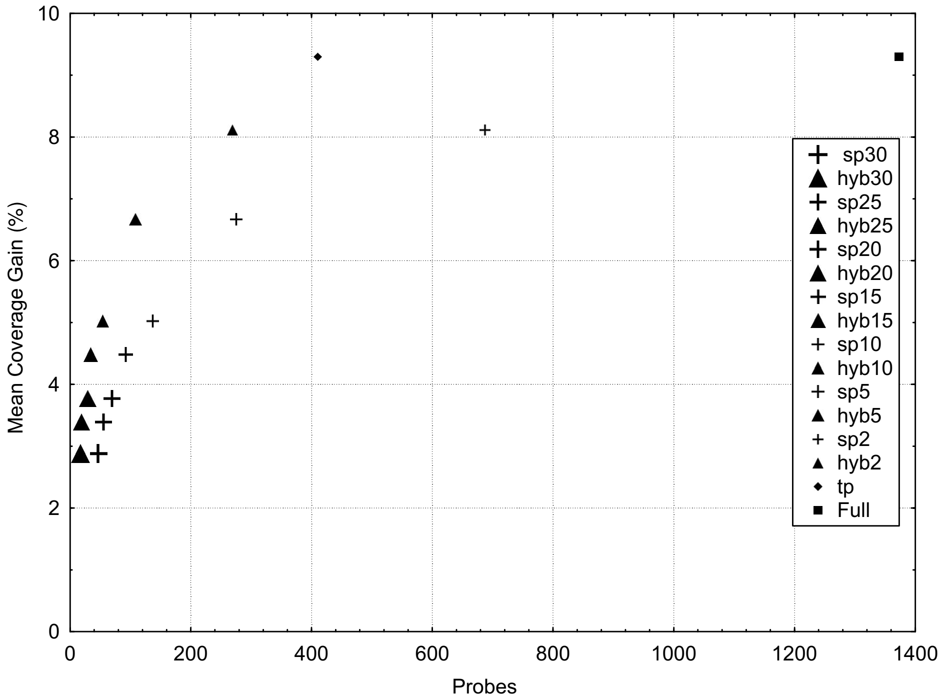

Figure 3: Coverage Gains and Probes for *full*, tp, sp, and hyb

Efficiency: number of probes executed in the field. The number of inserted probes is a reasonable surrogate for assessing the transparency of the profiling activities that is independent of the particular field activity. The reduction in the number of inserted probes, however, does not necessarily imply a proportional reduction in the number of probes executed in the field. So, to complement the assessment from Figure 3, we computed the reduction in the number of executed probes by analyzing the collected trace files.

Figure 4 presents the boxplots for the tp and sp profiling techniques. Tp shows an average overhead reduction of approximately 96%. The sp techniques' efficiency improves as the number of deployed versions increases (the number of probes for each user decreases), with an average reduction ranging from 52% for $sp2$ to 98% reduction in $sp30$. It is also interesting to note the larger efficiency variation of sp techniques when utilizing fewer versions; having fewer versions implies more probes per deployed version, which increases the chances of observing variations across users with different operational profiles.

Sampling across users and events. Figure 3 shows a considerable reduction in the number of probes through the family of stratified sampling techniques. As elaborated in Section 2.3, this

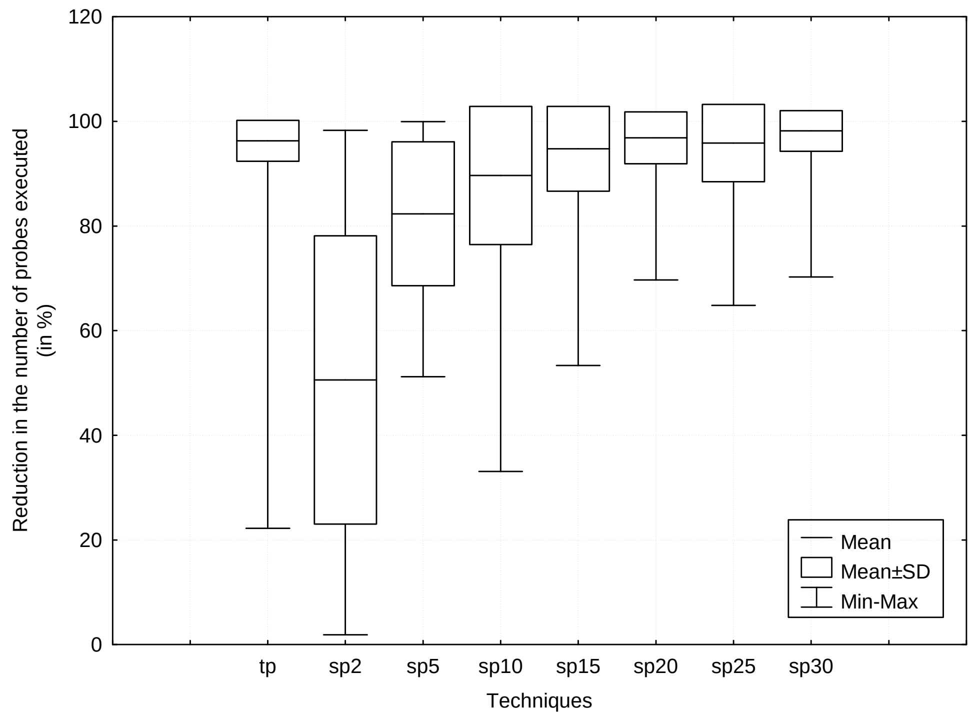

Figure 4: Executed Probes for tp, sp, and hyb

strategy adds a sampling dimension by distributing multiple versions with different sampled events (different instrumentation probes) across the user pool. We now investigate whether the reduction in the number of probes observed is due merely to the sampling across the pool of events, or whether sampling across the pool of users had a meaningful effect in the observed reduction.

Since we are interested in observing trends, we decided to focus on $sp2$ because it has the larger number of probes of all sp techniques. Under $sp2$, two versions are generated, each one with half of the total probes required by the $full$ technique. We re-label this technique as $sp2(50\%)$. We also created 5 variations of this technique where each of the two versions get a sample of the probes in $sp2(50\%)$: 80% results in $sp2(41\%)$, two-thirds results in $sp2(34\%)$, a half results in $sp2(25\%)$, a third results in $sp2(16\%)$, and a fifth results in $sp2(10\%)$. Then, we generated the analogous techniques that sample just across events within the same version, resulting in: $sp1(50\%)$, $sp1(41\%)$, $sp1(34\%)$, $sp1(25\%)$, $sp1(16\%)$, and $sp1(10\%)$. We performed this process for $sp1$ and $sp2$ 5 times, and computed the average additional coverage gain and number of probes inserted.

The results are presented in Figure 5. The gain decreases for both sets of techniques as the number of probes decreases. The figure also shows that adding the sampling dimension across users accounted for almost half of the gain at the 50% event sampling level. $Sp2$ was consistently more effective than $sp1$, even though the gap of effectiveness diminished as the number of probes was reduced. $Sp2$ provides a broader (more events are profiled over all deployed instances) but likely shallower (less observations per event) understanding of the event population than $sp1$. Still, as the number of deployed instances grows, $sp2$ is likely to compensate by gaining multiple observation points per event across all instances.

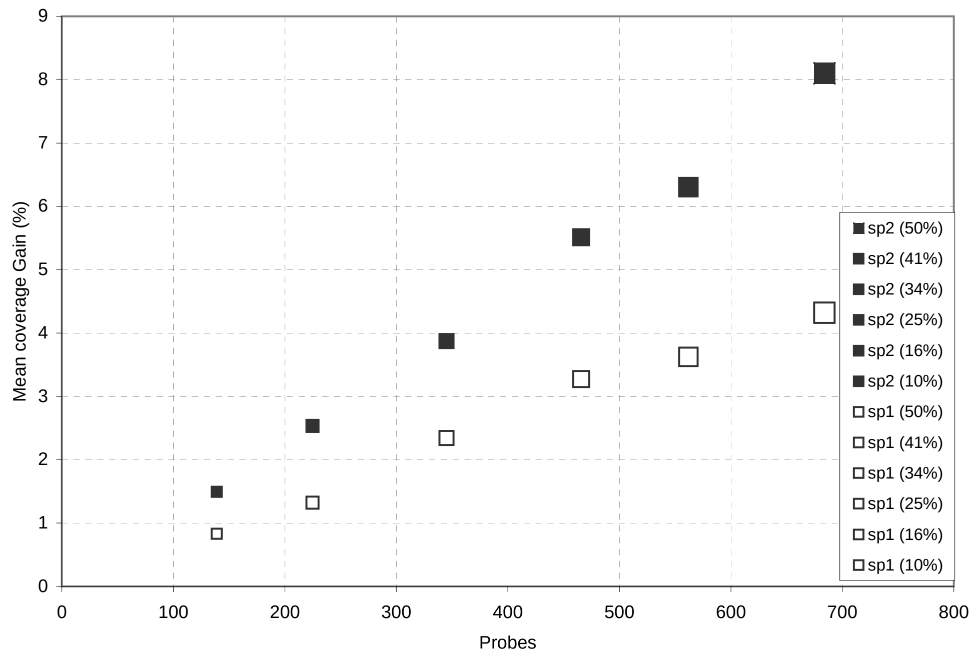

Figure 5: Sampling Dimensions: Gains for $sp1$ vs. $sp2$

Sampling and data loss. To get a better understanding of how much information is lost when we sample across deployed instances, we identified the top 5% (68) most executed functions under the *full* technique and compared it against the same list generated when various sampling schemes are used. We executed each sp technique five times to avoid potential sampling bias. The results are summarized in Table 5.

Utilizing $sp2$ reduces the number of probes in each deployed instance by half, but it can still identify 65 of the top 68 executed functions. When the sampling became more aggressive, data loss increased. The magnitude of the losses based on the sampling aggressiveness is clearly exemplified when $sp30$ is used and only 61% of the top 5% executed functions are correctly identified.

Interestingly, the number of probe reductions occurred at a higher rate than the data loss. For $sp30$, the number of probes was reduced to 3% when compared to *full* profiling. It was also noticeable that more aggressive sampling did not necessarily translate into higher standard deviations due to the particular sample. In other words, the results are quite independent from the functions that end up being sampled on each deployed version.

Overall, even the most extreme sampling across deployed instances did better than the in-house test suite in identifying the top executing functions. The in-house test suite, however, was not designed with this goal in mind. Still, it is interesting to realize that $sp30$, with just 45 probes per deployed version in a program with 1373 functions, can identify 61% of the most executed functions.

Table 5: Sp data loss versus *Full*

Technique	Probes	Top 5% Functions		
		Common Functions	%	Std.Dev
sp2	686	65	96	0.9
sp5	274	55	81	2.2
sp10	137	49	72	1.5
sp15	91	45	66	2.2
sp20	68	44	65	1.7
sp25	54	43	64	2.6
sp30	45	41	61	2.5
House	1373	35	51	-

4.3 RQ3 Study: Anomaly Based Triggers for Transfers

This study addresses RQ3, assessing the effect of detecting anomalies in program behavior to trigger data transfers. The objective is to trigger a field data transfer only when the released software is operating in ways we did not anticipate. We empirically evaluate various techniques and assess their impact in the coverage and fault detection capabilities of test suites generated based on field data.

4.3.1 Simulation Setting

We have concentrated on three families of anomaly-based detection techniques: departures from operational profile ranges, invariant violations, and unknown or failure-prone scenarios as defined by markov models. In our analysis we also include the full profiling technique, which triggers transfers irrespective of anomalies, providing a point of reference for comparison.

For each technique, we use the in-house test suite and a random group of users that simulate beta-testers to derive the nominal behavior against which field behavior is evaluated. We explore three levels of characterization for the baseline utilizing 3(10%), 6(20%), and 15(50%) of users.

When an anomaly is detected, the captured field behavior is sent to the server for the software engineers to determine how this anomaly should be treated (e.g., discard, re-assess baseline, reproduce scenario). For each technique we evaluate two variations: one that keeps $C_{houseTolerance}$ constant, and one that uses the information from the field to continually refine $C_{houseTolerance}$ (simulating the scenario in which engineers classify a behavior as normal and refine the baseline).

We now described the simulation settings for each technique individually.

Anomaly Detection Through Operational Profile Ranges. An operational profile consists of a set of operations and their associated probabilities of occurrences [23]. For this anomaly detection technique, the set of events to profile $C = O$, where O is the set of n operations o, and $C_{houseTolerance}$ can be defined by various ranges of acceptable execution probability for the operations. To develop the operational profile baseline for *Pine*, we followed the methodology describe by Musa [23]: (1) identify operations initiators; in our case these were the regular users and system controller; (2) create a list of operations by traversing the menus just as regular users would initiate actions; and (3) perform several pruning iterations, trimming operations that are not fit (e.g., operation that cannot exist in isolation). The final list contained 34 operations.

Although there are many ways to define $C_{houseTolerance}$ based on operational profiles, in this paper we just report on the technique that has given us the best cost-benefit so far: *Minmax* [6]. *Minmax* defines a range $[minimum, maximum]_o$, which contains the minimum and maximum execution probability per operation observed through the in-house testing process and the beta-testers usage. The technique assumes a vector with these ranges is embedded within each deployed

[6]For other techniques based on operational profiles see [6]).

instance. [7] Every time an operation has a field execution probability $field_o$, such that $field_o \ni [minimum, maximum]_o$, an anomaly is triggered.

Given the variable users' sessions duration, we decided to control this source of variation so that the number of operations in each session did not impact $field_o$ (e.g., sessions containing just one operation would assign 100% execution probability to that operation and most likely generate an outlier). The simulation considers partitions of 10 operations (mean number of operations across all sessions) for each user so that a session with less than 10 operations is clustered with the following session from the same deployed site until 10 operations are gathered, while partitioning longer sessions to obtain partitions of the same size.

Anomaly Detection Through Invariant Violation For this anomaly detection technique, $C = Inv$, where Inv is the set of program invariants inferred dynamically from the in-house test suite and the beta-testers, and $C_{houseTolerance} = 0$, that is, as soon as a invariant is violated a transfer is triggered.

Section 3.2 described how we inferred program invariants from the in-house test suite. To detect profile invariants in the field, such invariants could be incorporated into *Pine* in the form of assertions. When an assertion is violated in the field, then a transfer is triggered. Contrary to operational profiles, we did not have assertions inserted during the data collection process so we had to simulate that scenario. We employ the trace files generated from running the $M2$ test suite (the most powerful generation technique defined in Section 4.1) to approximate the trace files we would have observed in the field. Then, we analyze the invariant information generated from the traces to determine what original invariants were violated in the field.

Anomaly Detection Through Markov Models For this anomaly detection technique, we used a classification scheme based on markov models to characterize the field behavior into three groups: $pass$, $fail$, and $unknown$. When a field behavior is classified as either $fail$ or $unknown$ we trigger a transfer.

To create such a classifier, we followed the process utilized by Bowring et al. [19], which combines markov models to characterize program runs with clustering algorithms to group those runs (as performed by Dickinson et al [5]). From each event trace generated in-house and by the beta-testers, we created a markov model. Each model had an associated label indicating whether the

[7] Note that under Musa's approach only one operational profile is used to characterize the system behavior independent of the variation among users.

trace had resulted in a $pass$ or $fail$. We then used a standard agglomerative hierarchical clustering procedure to incrementally join these models based on their similarities. To assess the models' similarities, we transformed the models' probabilities to 0s and 1s (non-zero probabilities were considered 1) and computed their hamming distance. We experimented with various clustering stopping criteria and settled for 25% (two models cannot be joined if they differ in more than a quarter of their events). As a result of this process, the classifier of $pass$ behavior for functions had 67 models, and the one for operations had 53 models. Meanwhile, the classifier of $fail$ behavior for functions had 17 models and 5 models for operations.

We assume the classifiers can be consulted by the profiling process before deciding whether session should be transferred or not. Then, a field execution is associated with the model under which it has the greater probability to occur. If the markov model represents a $fail$ type behavior, then the execution trace is classified as exhibiting a $fail$ behavior. If the markov model represents a $pass$ type behavior, then the execution trace is classified as exhibiting a $pass$ behavior. However, if the field trace does not generate a probability above 0.1 for any model, then it is considered $unknown$.

In this study we consider traces of functions and operations. This lead to two classifiers $Markov_F$ and $Markov_{Op}$.

4.3.2 Results

Table 6 summarizes the findings of this study. The first column identifies the triggering technique we employed. Column two contains the number of beta testers (and the percentage over all users) considered to define the baseline behavior considered normal. Column three reports the number of transfers required under each combination of technique and number of beta testers. Columns four and five correspond to the number of functions covered and the new faults found when we employed the test suite generated with field data.

Independently of the anomaly detection technique, we observe some common trends. First, all anomaly detection techniques reduce, to different levels, the number of transfers required by the $full$ technique. Since an average of 250kb of compressed data was collected in each session, reducing the number of transfers had a considerable impact in the required packaging process and bandwidth. These technique also sacrifice, to different extent, the potential gains in coverage and fault detection. Second, having a better initial characterization of the system behavior through

Table 6: The effects of anomaly-based triggers

Technique	Beta Testers #	Beta Testers %	% Transfers	% Function Covered	% Additional. Functions Covered	Faults Found
Full	3	10	100	100	100	9
Minmax			28	97	83	1
Minmax+f			7	96	79	1
Inv			91	100	100	4
Inv+f			73	100	100	3
Markov$_F$			87	100	100	5
Markov$_{Op}$			64	97	86	3
Full	6	20	100	100	100	9
Minmax			12	98	93	2
Minmax+f			4	97	92	1
Inv			76	100	100	5
Inv+f			41	100	100	5
Markov$_F$			78	100	100	6
Markov$_{Op}$			41	98	89	5
Full	15	50	100	100	100	9
Minmax			2	97	77	2
Minmax+f			1	97	75	2
Inv			54	100	100	6
Inv+f			37	100	100	5
Markov$_F$			64	100	100	8
Markov$_{Op}$			22	99	85	5

additional beta-testers leads to a further reduction in the number of transfers. For example, with a characterization including just 3 beta testers, $minmax$ triggers 28% of the transfers required by $full$; when 15 beta testers are utilized, that percentage is reduced to 2%. Third, utilizing feedback reduces the number of transfers. For example, for the Inv technique and a characterization of 6 beta testers, adding feedback resulted in 35% less transfers. The cost of such reduction in terms of coverage and fault detection varies across techniques.

$Minmax$ is the technique that offers the greatest reduction in the number of transfers, requiring a minimum of 1% of the transfers utilized by $full$ (with 15 beta testers and feedback). The coverage obtained even with such small number of transfers is within 20% of $full$. However, a test suite utilizing the data from those transfers only detects 2 of 9 faults. The utilization of feedback increased $C_{houseTolerance}$ as new minimum and maximum execution probabilities for operations were observed in the field. Increasing $C_{houseTolerance}$ reduced the number of transfers, but its impact was less noticeable as more beta testers were added. Note also that the efficiency of $minmax_f$ will wear down as $C_{houseTolerance}$ increases.

Inv techniques are more conservative than $minmax$ in terms of transfer reduction, requiring 37% of the transfers employed by $full$ in the best case. However, a test suite driven by the transferred field data would result in equal coverage as $full$ [8] with over 50% of its fault detection capability. Another difference with $minmax$ is that Inv can leverage feedback to reduce transfers even with 15 beta testers (from 54% to 37% of the transfers). It is important to note that the overhead of this technique is proportional to the number of invariants incorporated into the program in the form of assertions; adjustments to the type of invariants considered is likely to affect the transfer rate.

$Markov$ techniques seem to offer better reduction capabilities that the Inv techniques, and still detect a greater number of faults. (Due to the lack of differences, we only present the results for techniques utilizing feedback.) For example, under the best potential characterization, $Markov_F$ needs 64% of the transfers required by $full$ sacrificing only 1 fault. It is also interesting to note that while technique $Markov_F$ performs worse than $Markov_{Op}$ in reducing the number of transfer, $Markov_F$ is generally more effective. Furthermore, per definition, $Markov_F$ is as good as $full$ in function coverage gains. We did observe, however, that $Markov_F$ efficiency is susceptible to functions with very high execution probabilities that skew the models. Higher observable abstractions like the operations in $Markov_O$ have more evenly distributed executions makes the probability scores more stable.

Overall, $minmax$ offers extreme reduction capabilities which would fit settings where data transfers are too costly, only a quick exploration of the field behavior is possible, or the number of deployed instances is large enough that there is a need to filter the amount of data to be later analyzed in-house. Inv offers a more detailed characterization of the program behavior, which results in additional reports of departure from nominal behavior, but also increased fault detection. $Markov$-based techniques provide a better combination of reduction and fault detection gains, but their on-the-fly computational complexity may not make them viable in all settings.

[8] Note that we simulated the invariant anomaly detection in the field utilizing $M2$, which had a maximum functional coverage of 838. Still, per definition, this technique would result in a transfer when a new function in executed because that would result in a new PP.

5 Conclusions and Future Work

We have presented a family of empirical studies to investigate several implementations of the profiling strategies and their impact on testing activities. *Our findings confirm the large potential of field data obtained through these profiling strategies to drive test suite improvements.* However, our studies also show that the gains depend on many factors including the quality of the in-house validation activities, the process to transform potential gains into factual gains, and the particular application of the profiling techniques. Furthermore, the potential gains tended to stabilize over time pointing to the need for adaptable techniques that re-adjust the type and number of profiled events.

Our results also indicate that *targeted and sampling profiling techniques can reduce the number of instrumentation probes by up to one order of magnitude, which greatly increases the viability of profiling released software.* When coverage is used as the selection criteria, tp provides a safe approach (it does not lose potential coverage gains) to reduce the overhead. However, practitioners cannot expect to notice an important overhead reduction with tp in the presence of a poor in-house test suite, and it may not be realistic to assume that practitioners know exactly what to target when other criteria are used. On the other hand, the sampling profiling techniques were efficient independently of the in-house test suite attributes, but at the cost of some effectiveness. Although practitioners can expect to overcome some of the sp effectiveness loss by profiling additional sites, the representativeness achieved by the sampling process is crucial to guide the insertion of probes across events and sites. It was also noticeable that the simple sequential aggregation of tp and sp techniques to create hyb provided further overhead reduction.

We also found that anomaly based triggers can be effective at reducing the number of data transfers, a potential trouble spot as the amount of data collected or the number of profiled sites increases. It is reasonable for practitioners utilizing *some of the simple techniques we implemented to reduce the number of transfers by an order of magnitude and still retain 92% of the coverage achieved by full profiling.* More complex techniques that attempt to predict the pass-fail behavior of software can obtain half of that transfer reduction and still provide 85% of the fault detection capability of $full$. Still, further efforts are necessary to characterize the appropriateness of the techniques under different scenarios and to study the adaptation of more accurate, even if expensive, anomaly detection techniques.

Profiling released software techniques still have to tackle many difficult problems. First, one of the most significant challenges we found through our studies was to define how to evaluate the applicability, cost, and benefit of profiling techniques for released software. Our empirical methodology provides a viable model combining observation with simulation to investigate these techniques. Nevertheless, we still need to explore how to evaluate these strategies at a larger scale and include additional assessment measures such as deployment costs and bandwidth requirements. Second, we have yet to exploit the full potential of field data. For example, our studies quantified simple mechanisms to transform field data into test cases. Given the appropriate data, these mechanisms could be refined to better approximate the behavior at deployed sites. Furthermore, in the presence of thousands of deployed instances, this process must happen continually, which introduces new challenges to filter, interpret, and aggregate a potentially overwhelming amount of data. Last, profiling deployed software raises issues not only about acceptable overhead but also about user's confidentiality and privacy. Future techniques and studies must take these factors into consideration.

Acknowledgments

This work was supported in part by a NSF-ITR program under award 0080898 and a CAREER Award 0347518 to University of Nebraska, Lincoln. We especially thank the users who volunteered for the study. We would also like to thank the participants of the First Workshop on Remote Analysis and Measurement of Software Systems, Gregg Rothermel and Michael Ernst for providing valuable feedback on earlier versions of this paper, and to the ISSTA 2004 reviewers for their suggestions to improve the paper.

References

[1] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[2] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[3] T. Ball and J. Laurus. Optimally profiling and tracing programs. In *Annual Symposium on Principles of Programming*, pages 59–70, August 1992.

[4] B. Calder, P. Feller, and A. Eustace. Value profiling. In *International Symposium on Microarchitecture*, pages 259–269, December 1997.

[5] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *International Conference on Software Engineering*, pages 339 – 348, May 2001.

[6] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *International Symposium of Software Testing and Analysis*, pages 65–75, July 2004.

[7] S. Elbaum, S. Kanduri, and A.Andrews. Anomalies as precursors of field failures. In *International Symposium of Software Reliability Engineering*, pages 108–118, 2003.

[8] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[9] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[10] A. Glenn, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, 1997.

[11] S. Graham and M. McKusick. Gprof: a call graph execution profiler. *ACM SIGPLAN*, 17(6):120–126, June 1982.

[12] K. Gross, S. McMaster, A. Porter, A. Urmanov, and L. Votta. Proactive system maintenance using software telemetry. In *Workshop on Remote Analysis and Monitoring Software Systems*, pages 24–26, 2003.

[13] MIT Program Analysis Group. The daikon invariant detector user manual. http://pag.csail.mit.edu/daikon/download/doc/daikon.html.

[14] M. Harrold, R. Lipton, and A. Orso. Gamma: Continuous evolution of software after deployment. cc.gatech.edu/aristotle/Research/Projects/gamma.html.

[15] D. Hilbert and D. Redmiles. An approach to large-scale collection of application usage data over the Internet. In *International Conference on Software Engineering*, pages 136–145, 1998.

[16] D. Hilbert and D. Redmiles. Separating the wheat from the chaff in internet-mediated user feedback, 1998.

[17] InCert. Rapid failure recovery to eliminate application downtime. www.incert.com, June 2001.

[18] J.Bowring, A. Orso, and M.J. Harrold. Monitoring deployed software using software tomography. In *Workshop on Program analysis for software tools and engineering*, pages 2–9, 2002.

[19] J.Bowring, J. Rehg, and M.J. Harrold. Active learning for automatic classification of software behavior. In *International Symposium of Software Testing and Analysis*, pages 195–205, 2004.

[20] D. Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, Inc., Sebastopol, CA, November 1996.

[21] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conference on Programming Language Design and Implementation*, pages 141–154. ACM, June 2003.

[22] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D.C. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *International Conference on Software Engineering*, pages 449–458, May 2004.

[23] J. Musa. *Software Reliability Engineering*. McGraw-Hill, New York, NY, 1999.

[24] Netscape. Netscape quality feedback system. home.netscape.com/communicator/navigator/v4.5/qfs1.html.

[25] Nielsen. Nielsen net ratings: Nearly 40 million Internet users connect via broadband. www.nielsen-netratings.com, 2003.

[26] University of Washington. Pine information center. http://www.washington.edu/pine/.

[27] A. Orso, T. Apiwattanapong, and M.J.Harrold. Leveraging field data for impact analysis and regression testing. In *Foundations of Software Engineering*, pages 128–137. ACM, September 2003.

[28] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *International Symposium on Software Testing and Analysis*, pages 65–69, 2002.

[29] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *International Conference of Software Engineering*, pages 277–284, May 1999.

[30] S. Reiss and M. Renieris. Encoding Program Executions. In *International Conference of Software Engineering*, pages 221–230, May 2001.

[31] D. Richardson, L. Clarke, L. Osterweil, and M. Young. Perpetual testing project. http://www.ics.uci.edu/ djr/edcs/PerpTest.html.

[32] A. van der Hoek, R. Hall, D. Heimbigner, and A. Wolf. Software release management. In M. Jazayeri and H. Schauer, editors, *European Software Engineering Conference*, pages 159–175. Springer–Verlag, 1997.

[33] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *International Symposium of Software Testing and Analysis*, pages 45 – 54, July 2004.

[34] C. Yilmaz, A. Porter, and A. Schmidt. Distributed continuous quality assurance: The Skoll project. In *Workshop on Remote Analysis and Monitoring Software Systems*, pages 16–19, 2003.