FAUSTA: Scaling Dynamic Analysis with Traffic Generation at WhatsApp

Ke Mao
Meta
kemao@fb.com

Timotej Kapus
Meta
kapust@fb.com

Lambros Petrou
Meta
petrou@fb.com

Ákos Hajdu
Meta
akoshajdu@fb.com

Matteo Marescotti
Meta
mmatteo@fb.com

Andreas Löscher
Meta
loscher@fb.com

Mark Harman
Meta
markharman@fb.com

Dino Distefano
Meta
ddino@fb.com

***Abstract*—We introduce FAUSTA, an algorithmic traffic generation platform that enables analysis and testing at scale. FAUSTA has been deployed at Meta to analyze and test the WhatsApp platform infrastructure since September 2020, enabling WhatsApp developers to deploy reliable code changes to a code base of millions of lines of code, supporting over 2 billion users who rely on WhatsApp for their daily communications. FAUSTA covers expected and unexpected program behaviors in a privacy-safe controlled environment to support multiple use cases such as reliability testing, privacy analysis and performance regression detection. It currently supports three different algorithmic input generation strategies, each of which construct realistic backend server traffic that closely simulates production data, without replaying *any* real user data. FAUSTA has been deployed and closely integrated into the WhatsApp continuous integration process, catching bugs in development before they hit production. We report on the development and deployment of FAUSTA's reliability use case between September 2020 and August 2021. During this period it has found 1,876 unique reliability issues, with a fix rate of 74%, indicating a high degree of true positive fault revelation. We also report on the distribution of fault types revealed by FAUSTA, and the correlation between coverage and faults found. Overall, we do find evidence that higher coverage is correlated with fault revelation.**

***Index Terms*—software testing, software reliability, dynamic analysis, continuous integration**

I. INTRODUCTION

Dynamic analysis and testing have been used to identify potential software defects in correctness, performance, security, and privacy [1]–[3]. Effective dynamic analysis requires a comprehensive set of realistic inputs. However, retaining or anonymizing production inputs for performing the analysis does not guarantee privacy. Our testing approach does not replay *any* real production data in order to remain privacy safe. Therefore, the input set needs to be generated. The generation process seeks to cover as many program behaviors as possible. In recent years, input generation techniques have been widely studied and practiced for mobile app testing and analysis [4]–[7], especially for the Android platform. As a result, the problem of testing client-side applications is relatively well understood. To study the equally important problem of server-side testing, this paper introduces the first report of industrial deployment of computational search algorithms that generate realistic simulated server-side traffic that has been deployed into continuous integration in industry at scale (applications consisting of millions of lines of code, used by over two billion users every day).

FAUSTA (**F**ully-**AU**tomated **S**erver **T**esting and **A**nalysis) was developed as a platform for next-generation server testing and analysis of WhatsApp, a well-known communication platform deployed by Meta. FAUSTA provides a framework for dynamic code analysis and testing with the end goal of helping developers gain confidence in their code changes at an early stage of the development cycle. The analysis focuses on helping developers to enforce software reliability, privacy, and performance for WhatsApp backend services, without relying on human engineers to write and maintain the tests themselves. FAUSTA is designed as a platform to allow back-end service owners to onboard their own products and use-cases.

We initially developed FAUSTA for improving code coverage and reducing test maintenance effort. At the time, test code coverage of the server code base was limited, even though there was a non-trivial number of unit and end-to-end tests implemented. Developers also suffered from debugging and maintaining flaky tests. While we encouraged developers to continue writing high-quality tests, we proposed to complement human effort via fully-automatic dynamic analysis. This analysis aims at comprehensive input generation, especially to cover those labor-intensive cases, and to reduce manual effort on maintaining tests. FAUSTA's input generation is based on specifications that capture templates of input patterns. In order to increase coverage we used three complementary algorithmic strategies: biased random generation, Markov-based generation [8] and evolutionary search techniques [9].

In order to achieve complete assured privacy safety (by construction), FAUSTA's design philosophy is to perform all its input generation and analysis without replaying real production traffic. Furthermore, the traffic FAUSTA generates does not contain real user data. This ensures that the FAUSTA traffic can be safely used in a development environment.

FAUSTA was firstly deployed to help improve service reliability. FAUSTA has also been deployed to generate traffic for privacy and performance regression detection. In this paper, we focus on the design, deployment and application of FAUSTA to the problem of testing server-side code for reliability in one

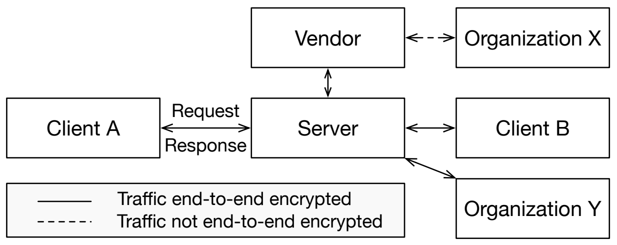

Fig. 1: An illustration of WhatsApp Server and its traffic [10].

of the world's largest communication platforms: WhatsApp.

The key contributions of this work are as follows: We introduce FAUSTA, a traffic generation platform that supports large scale testing and analysis. We present FAUSTA's traffic generation methodologies and our experience in increasing coverage with stochastic model-based strategies. We report on the industrial implementation and deployment of FAUSTA's reliability use case at WhatsApp to support testing server applications at a large scale. Specifically, we reveal that FAUSTA has found 1,876 unique reliability issues before production, with a 74% fix rate, since its deployment in September 2020. We report on the correlation between fault revelation and code coverage, and also give distribution data that indicates the types of faults found by this server-side testing approach. Taken together, these results provide baseline data on industrial deployment of server-side testing at scale, for future research and development.

II. BACKGROUND

In this section, we present an overview of traffic generation with an emphasis on testing and analysis. Moreover, we introduce traffic specifications used in WhatsApp client/server communications.

A. Overview of Traffic Generation

For providing effective dynamic analysis able to cover a comprehensive set of behaviors, FAUSTA needs to generate various types of traffic depending on the target use cases. For example:

- Realistic client traffic (that is similar to production) may be used to help ensuring critical user requests are handled as expected, or used to detect performance regressions.
- Unrealistic traffic (robustness testing) may be used to exercise corner-case behaviours and show crashes in the server or expose potential security risks.
- Synthesized traffic with artificial *Personally Identifiable Information* (PII) may be used to help dynamic analysis trace their propagation and logging and therefore prevent privacy regressions.

Such multi-purpose traffic generation requires deep understanding of the use cases and the service under test (SUT).

In this paper, our SUTs are from WhatsApp Server, which is composed of hundreds of Erlang service applications. Fig. 1 depicts WhatsApp Server and its traffic. Traffic is bi-directional between server and clients. Traffic could be end-to-end encrypted [10] or not end-to-end encrypted when routed via a *Vendor*. A user (*Client A*) may establish private connections (which are end-to-end encrypted) to another user (*Client B*) or organization (*Organization Y*). An organization may programmatically send and receive messages via server exposed APIs. An organization (*Organization Y* in Fig. 1) who uses these APIs may designate other infra provider as the *Vendor* to operate the API endpoints on their behalf. Note that some of the SUTs depend on non-Erlang endpoints at Meta such as the Business API endpoints [11].

WhatsApp server developers proactively detect reliability, security, and privacy risks by performing internal reviews, running code analyses, and using automated detection systems for risk identification and fixing. Externally, WhatsApp crowdsources the effort by engaging researchers through the Meta Bug Bounty Program [12] to improve the coverage of the detection of potential issues.

B. Traffic Specification

The traffic between clients and WhatsApp server follows a variation of XMPP [13] protocol. While the protocol itself is beyond the scope of this paper, it is important to note that messages between client and server can be described in XML. Listing 1 shows an example of such a message conveying latency info from an IP address. XMPP calls these messages *stanzas*. We have taken this example from Karpisek et. al.'s [14] work, which contains more details on reverse engineering the WhatsApp protocol.

Listing 1: A message sent to the server

```
<relayelection call-id="1431719979-2">
  <te latency="-98122">
     31.13.74.48:3478
  </te>
</relayelection>
```

According to RFC 6120, a server may opt to accept or send stanzas that have been validated against the specifications [15]. An example of a possible specification for the `relayelection` stanza in Listing 1 is shown in Listing 2.

Listing 2: A possible specification of message from Listing 1

```
<relayelection spec:call-id="id">
  <te spec:latency="int(x, y)">
     <spec:ip>
  </te>
</relayelection>
```

The specification is in XML and has a similar structure to the actual message. Tag and attribute names are the same, with the exception of the `spec` namespace. For example `<spec:ip>` denotes that an IP address with a port is expected in its place. Similarly `spec:latency="int(x,y)"` indicates that the `latency` attribute should be an integer between x and y (where x and y are instances of integers).

Note that this specification only helps validate individual stanzas and does not describe relationships between them. For example, it does not prevent an attempt to answer a call before anyone has made that call. However, this necessary context awareness between stanzas is addressed by separate dependency analysis phase described in Section III-D.

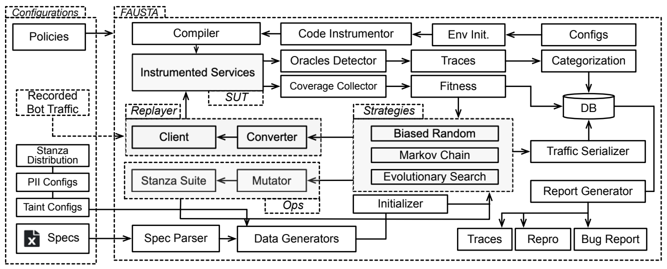

Fig. 2: FAUSTA's architecture.

III. Methodology Overview

A. Design

Fig. 2 shows FAUSTA's architecture. At Meta a 'diff' is the name given to a code change, while a task is the specification of a required code change or other action to be taken by a developer. The overall result from FAUSTA is therefore either to comment on a developer's diff (diff signals), or to file a task, outlining requirements for follow-up developer actions. The inputs are composed of three types of configurations:

1) Traffic specifications (specs) as described in Section II-B. These specs are relatively stable and available before the introduction of FAUSTA.
2) Specific use-case knowledge such as PII configs (which denote sensitive fields of the specs) and stanzas distribution. These are helpful in generating domain-specific traffic.[1]
3) Policies which are used to codify test oracles and instruct FAUSTA in the kind of violations to catch. The policies are use case specific. For example, for reliability testing, the oracle in this paper checks that the SUT should not crash, nor misbehave by producing soft errors.

FAUSTA takes these configurations as inputs. The *spec parser* parses traffic specs and sends them to *data generators* to initialize traffic. The *strategies* module optimizes the initialized traffic and sends it to a *replayer* which converts the generated traffic into a client readable format. The *replayer* sends traffic to *instrumented services*, which were compiled from automatically instrumented sources for coverage and stack trace profiling. Such *instrumented services* run in a non-prod, controlled environment to prevent the instrumentation from introducing any side effects to production. The *oracle detector* observes services under test to gather program behaviors including call stack *traces* for reporting errors. The *coverage collector* gathers line level coverage and uses evaluated traffic properties as *fitness* for feedback into *strategies*. Note that the *oracle detector* observes various types of oracles e.g., reliability, privacy and performance. A *report generator* collects the output and raises diff or task signals, with detailed bug report including traces and reproduction steps to help developers debug.

Next we describe how we generate traffic. Section III-B describes how we convert the specification into traffic. Section III-C presents Markovian and evolutionary strategies for improving the basic traffic generation.

[1]Note that an alternative traffic provision would be to record from internal bots and replay. However this is not the focus of this paper.

B. Traffic Generator

The specifications described in Section II-B allow us to generate individual stanzas. That is, we walk over the XML and replace all the elements or attribute values in the `spec` namespace with an instance of their type. We call this process *spec materialisation*. For example specification in Listing 2 could be materialised into the actual stanza in Listing 1.

We instantiate most types by picking a random value from the set the type represents. For example for `int(`x`, `y`)`, we could pick a random value between x and y. However we can influence the materialisation by making the type artificially stricter. For example in the case of `spec:call-id`, sometimes it would be desirable that some stanzas share the call id. If call ids were randomly generated that would be unlikely.

Algorithm 1 Traffic generation

```
1: $traffic \leftarrow \emptyset$
2: $ids \leftarrow$ GENERATEIDS()
3: while $spec \leftarrow$ PICKNEXTSPEC() do
4:     $stanza \leftarrow$ MATERIALISESPEC($spec$, $ids$)
5:     $traffic \leftarrow traffic \cup \{stanza\}$
6: end while
7: SENDTRAFFIC($traffic$)
```

Algorithm 1 describes the high-level structure of our traffic generator. The variable *traffic* is the set holding the generated traffic. It is first initialized to the empty set. Then the algorithm generates a small set of ids using the GENERATEIDS function that will be used in materialisation. Lines 3-7 show the main traffic generation loop. The loop iterates until it runs out of specifications to pick, represented by PICKNEXTSPEC function. There are different possible implementations of the function PICKNEXTSPEC. Our initial implementation PICKNEXTSPEC returns specifications in the order they are written on disk. While being a very simple strategy, it turned out that it is hard to beat. In Section III-C we discuss a more advanced picking strategy. The loop then materialises $stanzas$ using the $spec$ and ids using MATERIALISESPEC function, which walks the XML as presented before. Once the execution exits the loop, the generated traffic is sent to the server (line 7).

C. Stochastic Model based Generation

With stochastic model based generation we augment the basic Biased Random traffic generation with two different strategies for implementing the PICKNEXTSPEC function from Algorithm 1: a Markovian approach [8], based on predicting the next input and a computational search approach [9], based on evolving sequences of input.

Markovian Strategy: In this strategy we gather aggregated, sampled stanza data into a frequency distribution over specs and build a Markov chain. The goal of this strategy is to generate artificial traffic that has a similar profile to traffic observed in production. This is desirable because developers care more about errors that happen frequently in production.

The frequency distribution is obtained by matching incoming stanzas to the specification and increasing the count of that spec. The PICKNEXTSPEC function then samples this distribution a fixed number of times. That means the generated traffic has a similar profile to the production traffic. The most common stanzas seen in production are also the most common stanzas in generated traffic. The idea behind this approach is that the most common paths should also be tested more; a philosophy underpinning profile-based testing [16].

We then extended the frequency distribution approach by also building a Markov chain. States of the Markov chain represent our specifications. A transition between states $A \rightarrow B$ indicates that stanza fitting spec B is likely to follow a stanza fitting spec A. We build the Markov chain by mapping stanzas to specs and grouping them by hashes of users who sent them. For each hash we then order the stanzas by time at which they were received by the server. We then consider two consecutive stanzas: if they are close together in time we count it as a transition. Let us assume we observe the following traffic coming into the server from Alice starting at time $t + 0$:

Time	t + 0	t + 1	t + 2	t + 10	t+12
Spec	A	B	A	C	D

We would count transitions A $\rightarrow$ B, B $\rightarrow$ A and C $\rightarrow$ D, but not A $\rightarrow$ C, because they are too far apart in time.

This Markov chain is a dynamically built approximate model of the protocol. In other words, the Markov chain determines, given a stanza with spec A, the stanza that is most likely to be seen next.

We build the frequency distribution and the Markov chain continuously for traffic generation. First we sample an initial state from the frequency distribution. The initial state is then propagated through the Markov chain until it reaches a state with no outgoing transitions or has done a fixed number of propagations.

Evolutionary Strategy: FAUSTA also supports *evolutionary search*, which is capable of optimizing multiple conflictive objectives [17]. We have explored traffic minimization and coverage maximization via guided evolutionary search, which demonstrated convergences towards the favored properties of generated traffic. On fitness, we've implemented heuristics calculation to guide the search toward traffic minimization and coverage maximization. On representation, each individual in the population is a suite of stanzas, and the initial population is generated by randomly choosing from a pool of traffic specs and materializing them into stanzas. On operators, we perform a tournament selection, uniform crossover and whole suite mutation (*Mutator* in Fig. 2) by replacing the individual with new stanzas. At the time of writing, this strategy is currently under development and has yet to be fully deployed. Therefore, we do not elaborate further in this paper, and defer a detailed treatment for a future paper.

D. Guided Flows

Any approach based on stanzas needs to take account of inter-stanza dependence. FAUSTA does this using an approach we call 'guided flows'. Consider Alice tries to call Bob. First Alice sends a *MakeCall* stanza to Bob with a `call-id: 1235`. Then, to establish a call, Bob needs to send an *AnswerCall* stanza back to Alice with the same `call-id`. It is extremely unlikely that a random traffic generation approach such as that described in Section III-B would randomly materialise the same call-id for two otherwise independent stanzas *MakeCall* and *AnswerCall*. To tackle this, the algorithm needs context awareness, so that dependencies between stanzas can be taken into account when generating traffic.

Guided flows allow us to use developers' domain knowledge to help guide traffic generation. A guided flow consists of two parts. First a guided flow is just a sequence of specs to materialize. In our establishing a call example that would be *call_offer_spec* followed by a *call_answer_spec*, which indicates that a call answer stanza must be preceded by a call offer stanza. Second a guided flow needs a mechanism to share values between stanzas that depend on each other. To implement this mechanism, we leverage the ids from Algorithm 1. Our implementation is generic, and allows for arbitrary argument dependencies to be captured from developers' domain knowledge. In the remainder of this section we illustrate using the archetypal dependence of the call-id argument. Dynamically changing the ids between invocations to MATERIALISESPEC, gives us a mechanism to overwrite the value we care about to achieve the desired path. As an example of a guided flow, Listing 3 shows an Erlang sketch of a guided flow for establishing a call between Alice and Bob.

Listing 3: A guided flow example to establish a call

```
 1 CallId = 1235,
 2 OfferRandomIds = GenerateIds(),
 3 OfferIds = proplists:substitute_aliases(
 4   [{call_id, CallId}, {to, bob}, {from, alice}],
 5   OfferRandomIds),
 6 MaterialiseSpec(call_offer_spec, OfferIds),
 7 AnswerRandomIds = GenerateIds(),
 8 AnswerIds = proplists:substitute_aliases(
 9   [{call_id, CallId}, {to, alice}, {from, bob}],
10   AnswerRandomIds),
11 MaterialiseSpec(call_answer_spec, AnswerIds),
```

Listing 3 first generates some random ids at line 2. Then it overrides the *ids* we care about for the guided flow at lines 3-5. This makes sure that the stanza is sent from Alice to Bob and has a call id. Then the spec is materialised with these ids on line 6, for clarity we omit sending the materialised stanza in this exposition. The process is similar for the $AnswerCall$ stanza, with Bob sending a message to Alice using the same call id, which is what makes this flow valid. Notice that on line 7 we regenerates ids, meaning that all the ids that are not specifically overwritten, will have different random values. This means that guided flows retain some fuzzing capability.

IV. FAUSTA SYSTEM DEPLOYMENT

FAUSTA adopts a 'shift-left' software testing philosophy, which seeks to move testing effort early within the software development life-cycle. The shift-left strategy has been widely adopted by Meta for various types of issues e.g., reliability, performance, privacy and security. It has been used in testing client-side [5] as well as server-side (as reported here), and also has been used to test Meta's simulation platform, WW [18]. Meta's approach to the shift left philosophy is not restricted to testing, but has also been used in static analysis [19]. By shifting left, there are two major benefits:

Minimized negative impact: Early detection and fixing means reduced negative impact on users. When we find and fix bugs in the early stages of the development life-cycle, these bugs are removed before they reach production, and thereby leave users entirely unaffected; prevention is better than a cure.

Shorter time-to-fix: as is well known in software testing research [20], a defect revealed earlier in the life-cycle is less expensive to fix. This is particularly true in large-scale software deployment, such as WhatsApp at Meta: we have found that the necessary context switching imposed on developers by late life-cycle fault revelation significantly increases the cognitive burden on engineers [5], [19].

In the following we describe our practice in deploying FAUSTA for shifting left, with detailed workflows of the continuous integration (CI) pipeline.

A. CI Integration

FAUSTA has been integrated into WhatsApp CI pipeline for dynamic analysis and testing in multiple scenarios. Fig. 3 illustrates WhatsApp's software development life cycle and FAUSTA's role in this process:

Whenever a developer makes any changes of a subject system, the change set (aka the 'diff') needs to be submitted to a code review system. At Meta, this system is Phabricator, which provides a set of web-based collaborative software development tools such as code review and repository browser (Diffusion). The diff is required to be reviewed and stamped by at least one other engineer. On diff submission, a target determinator performs change impact analysis. It analyzes the diff changes and decides what CI jobs needs to be scheduled. For example, if a diff only touches comments/documentation and is no-op to production, a minimal set of CI jobs will be scheduled for saving computational resources. Test selection further analyzes what manually written test cases are associated with the diff changes and should be scheduled for checking functional correctness. Then a build job compiles the subject with diff changes and sends the built artifacts for testing and verification.

FAUSTA integrates into this process and performs dynamic code analysis for finding various types of issues e.g., reliability, privacy and performance. Other typical testing and verification jobs include static analysis, unit and end-to-end testing, and user (developer)-defined checker jobs etc. Once FAUSTA detects any issues, it performs fault localization based on collected stack trace to pin-point to a line that introduced the issue. This signal may get further boosted with other testing and verification job signals and get prioritized accordingly (e.g., a critical level signal will block developer's diff landing process). The common signal channels are diff signals (which appear as feedback of the diff in Phabricator) and tasks (usually triaged to the diff author or oncall of the subjects).

Based on FAUSTA's and other signals, the diff author may iterate above process multiple times until testing and verification signals are green. Together with a stamp from diff reviewer, developer can merge the diff into trunk. To verify the health of trunk, CI schedules continuous testing and verification jobs periodically, including FAUSTA's dynamic analysis. A healthy trunk revision triggers further continuous delivery jobs for releasing the changes to users.

B. Continuous and Diff Testing

CI jobs may have various schedules depending on the purposes. Two of the most common schedule types are continuous and diff. There are also other schedule types such as shadow jobs for verifying CI infra changes without affecting developers' user experience. A continuous job runs periodically on the 'trunk' version of the subject software repository, thereby checking overall system health, while multiple jobs also trigger on every developer's diff, as they are submitted for code review, thereby catching issues before they are even considered for merging into trunk. Both continuous and diff time deployments are necessary in practice. This is because diff time jobs may not cover issues only happen after merging multiple diffs into trunk (such as inter-diff dependences), while issues revealed by continuous testing usually require extra fault localization effort. To perform thorough dynamic analysis at an early stage, we've deployed FAUSTA supported reliability and privacy analyses for capturing risky changes. Its workflow works as follows:

Continuous detection workflow: A continuous FAUSTA run detects unexpected code behaviors on trunk/master.

(1) A scheduler triggers recurring dynamic analysis jobs for running FAUSTA hourly on master branch of the backend repository. (2) FAUSTA synthesizes, taints (for taint analysis that tracks how PII flows through the SUT), and tracks the newly generated traffic based on configs (e.g., error patterns for reliability and PII annotations for privacy). FAUSTA's taint tracing is supported by code instrumentation in the backend, specifically geared towards surfacing dynamic taint analysis results into call stacks. FAUSTA records its synthesized traffic to allow reproduction. (3) The dynamic analysis monitors code behaviors, where monitors differ per use case. For example, to help enforce privacy compliance, it tracks the PII prorogation and monitors how data sinks handle the data. To capture reliability issues, it monitors whether any service nodes crashed or suffered from soft errors. (4) A FAUSTA categorizer classifies unique issues and removes duplication. The categorization is based on stack trace analysis. (5) A FAUSTA configurable signal filter automatically reduces signals based on configs (e.g., it excludes known false positive cases). (6) The FAUSTA bot files tasks on newly detected issues. The report includes detailed stack traces, reproduction steps and subject metadata (e.g., repository commit hash) to help re-run and debug. This approach to tool integration in continuous deployment through software engineering bots has been widely studied in the sci-

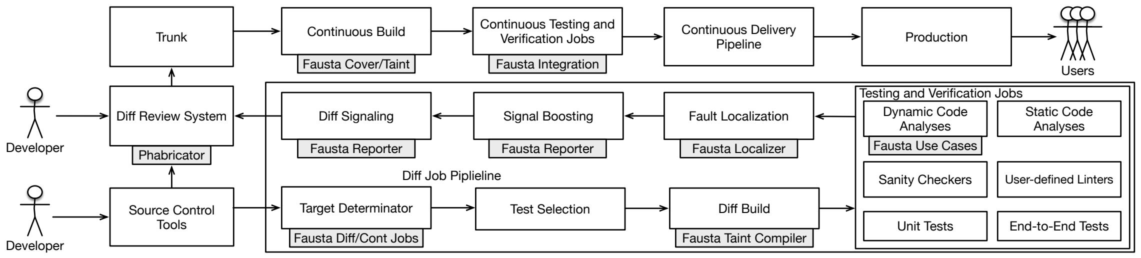

Fig. 3: FAUSTA's integration into Meta CI.

entific literature [21], and also deployed previously (and highly successfully) at Meta [5], [19]. (7) We use a 'human in the loop' approach to determine when to escalate risky issues on developer communication channels, notifying service owners and stakeholders. This is supported by the standard 'DevOps' software deployment methodology, widely used in industry, which ensures that there is always an available engineer on call with domain expertise [22]. (8) Service owners check or triage the reported issues and fix accordingly. Extra follow-up actions (e.g., retrospective for future prevention) may be required depending on the severity of the issues.

Diff-time detection: Diff-time signaling is one key step to further shift regression detection and fixing earlier. The workflow is similar to continuous detection, but the diff use case has stricter requirements in time cost of the analysis. The workflow needs to be efficient for reducing time-to-signal which is critical for developers' dev efficiency, as developers need to wait all diff job to finish before landing the changes, even though it's reviewed and accepted by another developer. Regarding signaling, instead of filing tasks, the issue reports are raised on developer's diffs via inline commenting in diff review tool (Phabricator). The Diff author then checks and fixes true positive cases before landing their diff (when it becomes merging into the repository trunk). For false positives, developers simply react to the FAUSTA signal as part of the review process. In this way, FAUSTA is a software engineering bot, giving its signal to developers as part of the code review process. Developers can also place reports in separate reporting channels for discussion and followup, thereby assisting the continuous improvement of the overall test process.

In practice, we found FAUSTA to be sufficiently efficient to finish from end to end when materializing traffic with all specs, without increasing developers' overall diff time waiting time. Fig. 4 shows the boxplot of FAUSTA diff time job cost, which contains statistics from 10,000 uniformly sampled FAUSTA jobs from the period between March 2021 and August 2021. On average, FAUSTA took 24.7 minutes per job. The best cases (1.5 times interquartile range) could finish in 11.3 minutes. FAUSTA could still finish in 38.7 minutes in most runs when we consider the worst cases[2]. These time performance results are comfortably within the implied Service Level Agreement (SLA[3]) required for FAUSTA to play the role of first responder

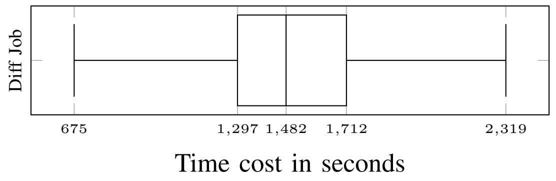

Fig. 4: End to end time cost of FAUSTA diff job (including diff build and harness setup time cost.

in the code review process, which is an important SLA for software testing bots [23].

C. Reporting Pipeline

Deploying dynamic analysis at scale not only requires solid technical foundations, but also good user experience. In addition to FAUSTA's dynamic analysis itself, FAUSTA aims to improve development efficiency by performing intelligent reporting with concise, reproducible and actionable signals. Its report pipeline contains the following main components:

Fault Categorization: The purpose of categorization is to classify unique issues, dedupe, and identify pre-existing ones. FAUSTA performs stack trace based categorization by choosing the top frame of the SUT for composing a categorization key. Note that the software classification problem itself is non-trivial and may frequently suffer from false grouping and false splitting issues. More sophisticated classification techniques are out of scope of this paper, which can be found in related studies [24], [25].

Fault Localization: When practicing automated testing and analysis techniques at scale in an industrial environment, one commonly reported issue is the heavy debugging effort, especially for end-to-end level input generation systems. To reduce our developers' debugging effort, FAUSTA fault localization pin-points to a line that caused the reported issue. The localization extracts stack traces from dynamic analysis logs, parses stack trace frames and locates a line that relates to the diff changes. Such localization info together with the call stack are reported as inline comments in Phabricator. For privacy use cases, the call stack instrumentation also helps developer understand which tainted data fields propagated to sensitive sinks, and how they were triggered. In addition, Fault localization on line level laid foundation for integrating with other automated fixing system at Meta e.g., SapFix [26], [27].

Signal Boosting: Signal boosting helps FAUSTA prioritize signals and filter out false positives. The basic signal boosting

[2]These were due to rare flaky infra failures such as networking issues caused retries. In such cases, no developer-visible signal was reported.

[3]Expected time-to-signal is set to 30 minutes at the time of writing.

uses a configurable, rule based approach to filter out false positives and overwrite issue priority based on empirical knowledge. We also analyze the error patterns detected by FAUSTA and down prioritize those that are less severe (e.g., soft errors). Signal boosting would also benefit from a combination of dynamic and static analysis in our existing CI pipeline. For example, if both existing static analyzers and FAUSTA dynamic analyzers pin point to the same issue, we may prioritize the issue and block developers from landing without fixing. Note that this work is still in progress and has not been deployed at the time of writing. Another related signal prioritization strategy that has been studied and practiced at industry is based on the prediction of which issues will likely get fixed by developers [28].

Fix Detection: To improve our understanding on the usefulness of FAUSTA reported signals, we've deployed fix detectors for tracking which diff reports got fixed and which did not. The fix detection is based on differential results across multiple versions of a diff, i.e., if FAUSTA reported an error on a certain version of a diff D, we check the existence of the error from D' subsequent versions (which also triggered FAUSTA diff jobs) to see whether the previously reported issue got fixed before merging into trunk. Note that this approach assumed that FAUSTA dynamic analysis is deterministic and reproducible. Such an assumption will not hold in real-world, complex systems, especially at the scales required for testing and a large tech deployment like Meta's. However, our analysis did show that FAUSTA detected issues are highly reproducible (and we also include the reproduction steps as part of FAUSTA reports). This allows us to achieve reasonable sufficiently good fix detection results as an approximation to true fixes.

V. Reliability Testing

This section presents results from the deployment of FAUSTA at Meta to test WhatsApp as a fully integrated testing technology within the company standard continuous integration system [29]. This continuous integration system is used daily by software engineers at Meta (including whom are specifically developing the WhatsApp platform). Results reported in the section have been collected over the full year from September 2020 to August 2021, thereby aggregating over a sufficient period of time to report reliable results for fault distributions, fault revelation and correlations between fault revelation and coverage.

Questions: We evaluate FAUSTA's reliability error detection capability to answer to following three questions:

- *Q1: Does FAUSTA find real world reliability errors with its generated traffic? Do developers fix them?*
- *Q2: What are the most common error types revealed by FAUSTA?*
- *Q3: Does coverage improvement lead to more unique errors detected?*

To answer the questions, we continuously collected data from the deployment of FAUSTA, including errors detected, fixes applied, coverage achieved and computational resources deployed to test, during the period between September 2020 and August 2021. We checked developers' reactions to the reports by running automated fix detection complemented by semi-automated manual analysis.

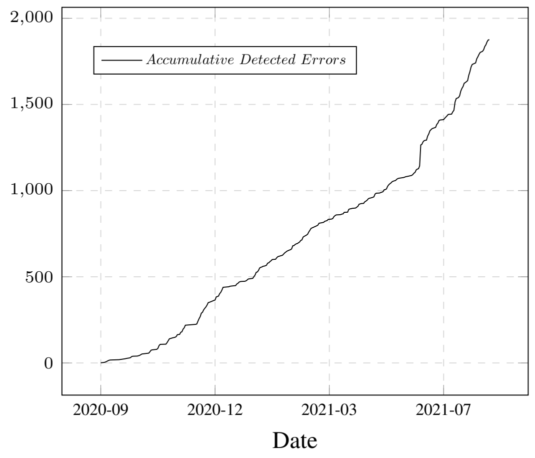

Fig. 5: Accumulative reliability errors detected by FAUSTA.

Fully automated fix detection within a continuous integration environment is a challenging research topic in its own right. Space would not permit us cover this topic in detail in the present paper. However, a more detailed treatment of the challenges of fully automated fix detection can be found elsewhere [5].

To understand the correlation between coverage and error detection capabilities, we instrumented the WhatsApp server backend, collecting data on coverage over millions of lines of Erlang code. This provides us with some of the largest scale empirical data on correlation between coverage and faults revealed in the literature, and certainly the largest to report on system level coverage correlations for Erlang. The closest related work is the recent report on unit test coverage at Ericsson [30], which found very little evidence for correlation between unit test coverage and fault revelation.

The WhatsApp backend also depends on other non-Erlang endpoints at Meta such as the Business API endpoints. In this reliability use case, we focus on the traffic between WhatsApp Erlang services and WhatsApp clients. In future work, we will develop and report upon traffic flows from WhatsApp server to Meta endpoints, together with other FAUSTA use cases to performance analysis and privacy assurance.

A. Detected Errors

A natural starting point for any study of software testing centres on the number of faults found by the testing approach. In this section we report the reliability errors detected by FAUSTA. Fig. 5 illustrates accumulative reliability errors detected by FAUSTA, between September 2020 and August 2021. In total, FAUSTA found 1,876 unique errors. On average, FAUSTA detected 7.9 unique reliability errors per work day.

The step changes in rate of fault detection seen in the figure are the result of the FAUSTA engineers feeding back results and refinement of the stanzas and input generation algorithms through the first year of deployment. In any industrial deployment of a software testing technology this feedback loop is vital to ensure continuous improvement (of testing workflows, fault reporting, coverage and fault revelation achieved).

Much of the software testing literature concerns the problem of false positives [20], [31]. Naturally, it is insufficient to report fault revelation data without also reporting on the number of faults deemed to be false positives. Of course, answering this question unequivocally would be equivalent to solving the halting problem, and so there can be no automated analysis that determines the *real* underlying false positive rate.

We do have high confidence in the realism of the test cases, and since the product we are testing has over 2 billion users, the probability that an issue detected by FAUSTA is a real false positive is low. However, as we have discovered in previous work on static analysis testing at Meta [5], [32], [33] the real false positive rate is *not* the primary concern. Some faults are possible in practice, yet so unlikely to occur, with so little chance of affecting real end users, that they become naturally de-prioritised in the fault fixing process. This leads to the concept of a pseudo-false positive [5].

A pseudo false positive is a fault report that may be a real fault, but which is not acted upon by developers because, in their expert judgement, the fault is low priority. The pseudo-false positive set contains the real false positive set. Therefore, assessing test effectiveness based on pseudo false positives is a more demanding evaluation criterion than assessing based on real false positives. It is also more closely related to the real world effectiveness of software testing. That is, a software testing technology that reports only true positives that are insufficiently high priority for any to be fixed is just as useless as a testing technology that reports only false positives. Therefore we focus on the *developer fix rate* [33] as our fundamental measure of test technology impact. The developer fix rate also serves as basis for determining the derived measurement of the number of pseudo-false positives.

Regarding developer fixes, we found a high fix rate on FAUSTA raised issues at code review time of 74%. We also found a much lower fix rate on FAUSTA detected issues from continuous runs of 20%[4], further underlining the importance of the overall shift left philosophy.[5]

In order to understand the reason for the 26% of issues raised that were not fixed, we performed a manual analysis, together with follow-up interviews with developers to understand the reasons why they may have chosen not to implement a fix, in response to an issue raised by FAUSTA. We expected the most common reason to turn out to be the observation that many of these issues were pseudo-false positives, and therefore de-prioritised. However our findings from this follow-up human analysis surprised us. Although there are pseudo-false positives among the 26%, we found the most common reason to not fix was because of the conflict between the system specs and human intention to 'let it crash' in Erlang (a development paradigm for enhancing fault tolerance).

[4]This figure is based on relatively small sample: 2 fixes from 10 issues reported to developers. By far the majority of the 1,876 faults reported in this paper were at code review time, as a result of shifting left.

[5]This confirms other experiences of industrial deployment of static analysis techniques such as those reported by Distefano et al. [19].

There is very little research in the literature on testing automated service for reliability at scale, and therefore we have no baseline against which to benchmark our developer fix rate. The closest published data we could find available, that is also comparable to our results reported here, was previous work on client-side automated testing [5]. Comparing against this previous work, we found that FAUSTA exhibits a very similar level of fix rate to Sapienz [5].

This is encouraging, because it is much easier to generate realistic test inputs on the client side; the GUI is available to distinguish between realisable input sequences and infeasible sequences. By contrast, without GUI guidance on the server side, it is possible to generate infeasible sequences. Research on automated testing has shown this to be one highly impactful reason for real false positives in automated testing technologies [34]. It was therefore encouraging that we could achieve similar fix rates on the server side, as those we were able to achieve on the client side.

B. The Distribution of Fault Types Revealed by FAUSTA

TABLE I: Error type distribution with exit reasons [35]

Reason	**Pct.**	**Description**
function_clause	53.63%	Missing matching function clause when evaluating a function call.
case_clause	19.29%	Missing matching branch when evaluating a case expression.
crash report	14.29%	Erlang crash report [36]
stack trace (soft error)	8.81%	Soft error with stack trace logged by developers.
badarg	1.77%	The argument is of wrong data type or badly formed.
badrecord	1.35%	A bad Erlang record data was found in an Erlang module.
exit	0.86%	The process called exit/1.
emulator error	0.01%	Erlang emulator error.
internal error	<0.01%	Thrift runtime internal error.

One of the unique contributions of industrial experience papers on software testing lies in the reporting of fault distribution data from closed source industrial production. Much academic software testing research necessarily involves open source software, or laboratory controlled experiments. This is because it is hard for academic researchers to gain access to closed source production industrial systems for evaluation of testing technologies. It is therefore scientifically valuable to be able to compare fault distribution data from industrial experience papers with fault distribution data from controlled academic experiments and empirical results based on open source systems. The scientific literature also, in turn, feeds back into industrial decision-making and development prioritisation. For example, our results from industrial deployment of Sapienz [37] allowed us to compare distributions of faults on Meta systems with previous baseline data from academic research on multiple different Android test generation systems [38] as well as our own previous scientific research, conducted while in academia, on the research prototype of Sapienz [39], [40]. Interestingly, the fault distribution categories and their prevalence proved to be surprisingly similar, suggesting that industrialists could benefit from academic research and vice versa.

In order to facilitate future comparison from industrial experience, and also this important academic scholarship on software testing, we report, in Table I, the distribution of types of fault revealed by FAUSTA. Table I presents the most common reliability error reasons revealed by FAUSTA. The top two most prevalent error types are related to Erlang's pattern matching mechanism, where a left-hand side pattern is expected to match a right-hand side term. At least 73% reliability issues were caused by missing patterns on either function or case level. Partial pattern mismatch issues in theory could be revealed by the static analysis tool Dialyzer [41], which is a popular Erlang tool for finding code discrepancies such as definite type errors. However, in practice we found it is common that FAUSTA's dynamic analysis signals are uniquely uncovered by FAUSTA, which may be attributed to Erlang's dynamic typing feature and its 'let it crash' philosophy. Note that Table I also includes general error reports which are uncategorized by root cases. For example, the 14% crash reports are general Erlang crash reports which could be triggered by pattern mismatch, while the stack trace reports contain soft errors handled and logged by developers.

Naturally, care has to be taken interpreting these results. Our results cannot indicate the (unknowable) distribution of faults in the code base under test, but only the prevalence of faults amongst those found by FAUSTA, which may be different. Furthermore, it is evident from the most prevalent four categories, that our results are highly specific to the programming language Erlang.

While this makes it harder to use the results as a baseline for future research on testing of other languages, for example backend systems implemented in Hack, the findings may, nevertheless be of interest to researchers in functional programming. Erlang is widely regarded as one industrial example of functional language deployment: real-world large-scale backend systems have been deployed, not only at Meta, but also notably at Ericsson. These results may thus contribute to scientific research (and practitioner discussion) on the correlation between programming language styles and prevalence of faults [42]–[44].

C. Coverage and Errors

Code coverage is a key foundation for analysis and testing. There has been much scientific study, and indeed considerable controversy, concerning the question as to whether code coverage is correlated with fault revelation [45]–[48]. Much of the literature has concerned relatively small programs, research prototypes, and/or open source systems. More empirical data is therefore required on large-scale closed source industrial systems in order to complete the empirical scientific analysis of these fundamental questions for software testing.

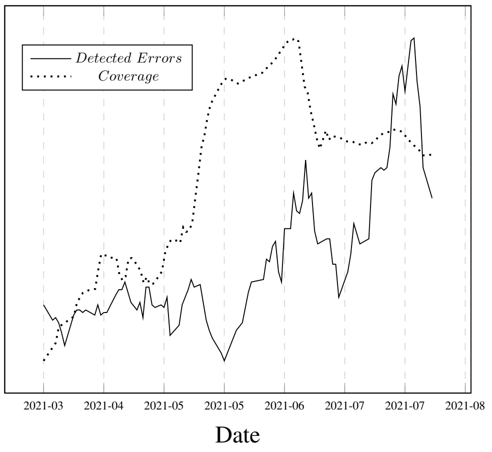

Fig. 6: Correlation between coverage and fault revelation.

In order to make our own, modest, contribution to this large and important research literature, we report the results from our own analysis of the correlation between FAUSTA code coverage and fault revelation on WhatsApp. Throughout the deployment of FAUSTA, we used feedback from CI production to inform testing techniques that lead to gradual increase in code coverage (see Section III). This gives us the ability to explore the correlation between coverage and fault revelation.

The closest study of correlation between coverage and fault revelation to ours is probably the previous results by Antinyan et al. [30] who reported coverage results from unit testing on a large-scale real-world telecoms project of approximately 2 million lines of code (LoC), under development by a team of approximately 150 engineers, with several major releases per year. These results are comparable to ours in sense that the number of developers and the size of the system is similar. Also, although it is never stated in their paper, it is not unlikely that the system also involved a significant amount of Erlang code. On the other hand, the results reported by Antinyan et al. concerned unit testing, whereas our results are for system level testing. Another key difference may reside in the observation that the Ericsson product is released publicly only occasionally, by comparison with WhatsApp, which is subject of continuous release. The most important difference, however, is probably the test generation approach: it is likely that many of the unit tests may have been written by human developers rather than an automated approach, such as FAUSTA.

In the previous study at Ericsson on unit testing [30], the evidence that coverage is correlated to fault revelation is extremely modest. By contrast we found stronger evidence for a correlation than this previous study on unit testing. That is, while Ericsson's unit testing paper reported correlation coefficients of approximately 0.2 (on a similar size communications system), we found coefficients of approximately 0.4 (for system level testing using FAUSTA).

Recent research on coverage-fault correlations, more generally, has suggested that the correlation, where it does exist, tends to exist only once relatively high levels of coverage have been achieved [48], and therefore we certainly do not dismiss the possibility that better fault revelation could be achieved by further improvements in code coverage. Furthermore, in order to gain confidence in the correctness of the system under test, we clearly need the highest possible code coverage, even if higher coverage were ultimately to prove uncorrelated with fault revelation. For both reasons, together with the modest correlation we have served so far, we do intend to consider and develop techniques to improve code coverage in our future work.

More specifically, to better understand the relationship between coverage and error detection capability, we collected FAUSTA daily detected errors together with its corresponding code coverage within the subject repository, for the period between March 2021 and August 2021. Fig. 6 shows the results, with 7-day moving average to smooth variations introduced by non workdays/weekends. Overall, we found that higher code coverage *did* correlate to FAUSTA catching more unique reliability errors, although both rank and linear correlations are relatively modest.

That is, the correlation between code line coverage and number of detected unique errors exhibits a Pearson correlation coefficient of 0.368 ($p = 0.0003$) and a Spearman rank correlation coefficient of 0.38 ($p = 0.0002$). In both cases the p value gives strong confidence that each of these two correlation coefficients is highly unlikely to be zero. The linear and rank based correlation coefficients are similar, and give evidence that there is a non-trivial, although modest, correlation between coverage and fault revelation. This is a further encouragement to us to consider techniques that may help us to further elevate server-side code coverage.

Finally we performed an end-to-end experiment to assess the impact of coverage improvement on the amount of errors found. For the whole month of June 2021 we ran two versions of FAUSTA on every published diff in our codebase. The first version was the latest version with coverage improvements we have implemented in 2021. The second version was an older version from the start of 2021. The latest version covered 2.27x more lines of code than the older version.

We then compared the errors reported on a diff between the old and new version of FAUSTA. There were 34 occurrences of issues reported by the new version of FAUSTA, that would have been missed if we were using the older version with lower coverage. There were 86 issues reported on diffs in that time period, which means the additional coverage contributed 39.5% of those issues.

VI. Related Work

This section presents related work on code analysis and input generation. There are several related code analysis and input generation systems already deployed at Meta. For example, Infer static analysis at scale [19], [33], Sapienz automated mobile testing with GUI exploration [5], and WES user interaction simulation [49]. These systems have demonstrated their effectiveness in hunting software defects under industrial settings. FAUSTA differs from these prior systems and is unique in empowering dynamic analysis for server applications via traffic generation.

On input generation, many techniques from academic and industrial communities have been proposed in the past decades [50]–[56]. Android client is one of the hottest subjects that has been massively studied on its test input generation. However, the problem remains challenging and largely unsolved in terms of coverage and benchmarking against random approaches [38]. For Web client, Wassermann et al. [57] proposed an algorithm to analyze dynamic code and to model its semantics based on runtime values. Artzi et al. [58] combined concrete and symbolic execution to generate inputs directly for dynamically-generated Web apps. Halfond et al. [59] improved test input generation via a novel static analysis algorithm to help discover Web interfaces. There are also a body of literatures that studied generation strategies for increasing coverage. Pacheco et al. [60] proposed a random test generation approach by using feedback from execution of the generated inputs. Pandita et al. [61] presented a general coverage approach for improving logical and boundary-value coverage. Wang et al. [62] proposed SAFL which uses symbolic execution and guided fuzzing for increasing test coverage.

VII. Conclusion and Future Work

This paper presents a system named FAUSTA for dynamic code analysis and testing at large scale with traffic generation. The FAUSTA approach and its industrial implementation (i.e., the FAUSTA system) have been successfully adopted to support two use cases on reliability testing and privacy analysis at WhatsApp. The system synthesizes various types of service inputs, injects them into a controlled sandbox environment and tracks their prorogation together with statement coverage to help analyze program behaviors. Its seamless integration into Meta CI supports developers in revealing software defects at an early stage of software development life cycle. Results from the reliability use case showed that the system helped prevent thousands of risky issues before reaching production.

In the future, we plan to further improve our traffic generation strategies (e.g., with new guided/targeted mechanisms) to cover a more comprehensive set of program behaviors. In addition to reliability testing, we plan to report our experience and lessons learned from the deployment of privacy and performance use cases. Another interesting area that we plan to explore is to combine dynamic analysis with other static analysis at Meta for improving software defect detection capability (especially in finding false negatives and reducing false positives).

Acknowledgements

The authors would like to acknowledge WhatsApp and Meta engineering leadership for supporting this work. The authors would also like to thank their colleagues who have provided useful feedback on the FAUSTA system.

References

[1] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–36, 2017.

[2] M. Tran, X. Dong, Z. Liang, and X. Jiang, "Tracking the trackers: Fast and scalable dynamic analysis of web content for privacy violations," in *International Conference on Applied Cryptography and Network Security*. Springer, 2012, pp. 418–435.

[3] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp. 204–217.

[4] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proc. of ISSTA'16*, 2016, pp. 94–105.

[5] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with Sapienz at Facebook," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 3–45.

[6] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 738–748.

[7] T. Cai, Z. Zhang, and P. Yang, "Fastbot: A multi-agent model-based test generation system beijing bytedance network technology co., ltd." in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, pp. 93–96.

[8] J. R. Norris, *Markov chains*. Cambridge university press, 1998, no. 2.

[9] M. Harman, P. McMinn, J. Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification: LASER 2009-2010*, B. Meyer and M. Nordio, Eds., 2012, pp. 1–59, LNCS 7007.

[10] "WhatsApp security whitepaper." [Online]. Available: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf

[11] "Whatsapp Business API." [Online]. Available: https://developers.facebook.com/docs/whatsapp

[12] "Facebook bug bounty program." [Online]. Available: https://www.facebook.com/whitehat

[13] "Uses of XMPP: Instant messaging." [Online]. Available: https://xmpp.org/uses/instant-messaging

[14] F. Karpisek, I. Baggili, and F. Breitinger, "Whatsapp network forensics: Decrypting and understanding the whatsapp call signaling messages," *Digital Investigation*, vol. 15, pp. 110–118, 2015, special Issue: Big Data and Intelligent Data Analysis.

[15] "RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core." [Online]. Available: https://xmpp.org/rfcs/rfc6120.html

[16] N. Li and Y. K. Malaiya, "On input profile selection for software testing," in *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*. IEEE, 1994, pp. 196–205.

[17] M. Harman, "The current state and future of search based software engineering," in *Proc. of FOSE'07*, 2007, pp. 342–357.

[18] J. Ahlgren, M. E. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, E. Meijer *et al.*, "Testing web enabled simulation at scale using metamorphic testing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 140–149.

[19] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at Facebook," *Communications of the ACM*, vol. 62, no. 8, pp. 62–70, 2019.

[20] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[21] M.-A. Storey and A. Zagalsky, "Disrupting developer productivity one bot at a time," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 928–931.

[22] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of devops concepts and challenges," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–35, 2019.

[23] K. Bojarczuk, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, S. Lucas, E. Meijer, R. Rojas, and S. Sapora, "Measurement challenges for cyber cyber digital twins: Experiences from the deployment of facebook's WW simulation system," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '21)*, October 2021.

[24] Y. C. Cavalcanti, E. S. de Almeida, C. E. A. da Cunha, D. Lucrédio, and S. R. de Lemos Meira, "An initial study on the bug report duplication problem," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 264–267.

[25] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems," in *2007 IEEE 23rd International Conference on Data Engineering Workshop*. IEEE, 2007, pp. 433–441.

[26] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[27] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 269–278.

[28] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 495–504.

[29] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.

[30] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron, "Mythical unit test coverage," *IEEE Software*, vol. 35, no. 3, pp. 73–79, 2018.

[31] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[32] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 1–23.

[33] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods Symposium*. Springer, 2015, pp. 3–11.

[34] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: high coverage, no false alarms," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 67–77.

[35] "Erlang errors and error handling." [Online]. Available: https://erlang.org/doc/reference_manual/errors.html#exit-reasons

[36] "Erlang SASL error logging." [Online]. Available: https://erlang.org/doc/apps/sasl/error_logging.html#crash-report

[37] "Friction-free fault-finding with Sapienz." [Online]. Available: https://developers.facebook.com/videos/f8-2018/friction-free-fault-finding-with-sapienz/

[38] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proc. of ASE'15*, 2015, pp. 429–440.

[39] K. Mao, "Multi-objective search-based mobile testing," Ph.D. dissertation, UCL (University College London), 2017.

[40] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[41] T. Lindahl and K. Sagonas, "Detecting software defects in telecom applications through lightweight static analysis: A war story," in *Asian Symposium on Programming Languages and Systems*. Springer, 2004, pp. 91–106.

[42] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek, "On the impact of programming languages on code quality: a reproduction study," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 4, pp. 1–24, 2019.

[43] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 155–165.

[44] J. Zhang, F. Li, D. Hao, M. Wang, H. Tang, L. Zhang, and M. Harman, "A study of bug resolution characteristics in popular programming languages," *IEEE Transactions on Software Engineering*, 2019.

[45] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 560–564.

[46] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *2009 3rd international symposium on empirical software engineering and measurement*. IEEE, 2009, pp. 291–301.

[47] L. Briand and D. Pfahl, "Using simulation for assessing the real impact of test coverage on defect coverage," in *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR00443)*. IEEE, 1999, pp. 148–157.

[48] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 597–608.

[49] J. Ahlgren, M. E. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, R. Laemmel, E. Meijer *et al.*, "WES: Agent-based user interaction simulation on real infrastructure," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 276–284.

[50] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004, pp. 97–107.

[51] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[52] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[53] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 987–992.

[54] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software." Vancouver, BC: USENIX Association, Aug. 2017.

[55] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[56] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[57] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proc. of ISSTA'08*, 2008, pp. 249–260.

[58] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proc. of ISSTA'08*, 2008, pp. 261–272.

[59] W. G. J. Halfond and A. Orso, "Improving test case generation for web applications using automated interface discovery," in *Proc. of FSE'07*, 2007, pp. 145–154.

[60] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 75–84.

[61] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," in *Proc. 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, September 2010.

[62] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 61–64.