# Automated Software Testing as a Service

George Candea
School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland
george.candea@epfl.ch

Stefan Bucur
School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland
stefan.bucur@epfl.ch

Cristian Zamfir
School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland
cristian.zamfir@epfl.ch

## ABSTRACT

This paper makes the case for TaaS—automated software testing as a cloud-based service. We present three kinds of TaaS: a "programmer's sidekick" enabling developers to thoroughly and promptly test their code with minimal upfront resource investment; a "home edition" on-demand testing service for consumers to verify the software they are about to install on their PC or mobile device; and a public "certification service," akin to Underwriters Labs, that independently assesses the reliability, safety, and security of software.

TaaS *automatically* tests software, without human involvement from the service user's or provider's side. This is unlike today's "testing as a service" businesses, which employ humans to write tests. Our goal is to take recently proposed techniques for automated testing—even if usable only on toy programs—and make them practical by modifying them to harness the resources of compute clouds. Preliminary work suggests it is technically feasible to do so, and we find that TaaS is also compelling from a social and business point of view.

**Categories and Subject Descriptors**
D.2.5 [*Testing and Debugging*]: Testing tools

**General Terms**: Reliability

## 1. INTRODUCTION

Software quality assurance is in dire need of substantial progress. Software testing is resource-hungry, time-consuming, labor-intensive, and prone to human omission and error. Despite massive investments in quality assurance, serious code defects are routinely discovered after software has been released [16], and fixing them at so late a stage carries substantial cost [13]. Thorough testing of large, complex software involves great effort, and the software industry still employs relatively primitive testing techniques.

The current software business model forces software users to take on faith that the vendor has performed thorough testing before shipping. Yet, given the difficulty of thoroughly testing software, such trust is typically misplaced. There exists no objective way to assess the reliability of a software product, therefore the main competitive metric is often performance and functionality. There is no independent certification body to guarantee that every vendor employs state-of-the-art testing.

We need a "disruptive technology" to substantially improve software quality. Various studies have found the average bug density in production-ready software to have stayed relatively constant over time, while average code volume of software has increased along an exponential curve [13], with the net effect that the number of bugs per product is increasing. It is therefore necessary to quickly find a way of reducing bug density by at least an order of magnitude. A promising direction is to reduce reliance on human labor through automated testing techniques, and recent proposals [4, 5, 2, 1] have made promising progress along these lines. Alas, they are still not ready to handle real-sized software (1 million lines of code or more), mainly due to high CPU and memory requirements. We believe cloud computing can come to the rescue.

### *The Promise of Automated Testing as a Service (TaaS)*

Software testing essentially consists of exercising as many paths through a program as possible and checking that certain properties hold along those paths (no crashes, no buffer overflows, etc.)

TaaS combines two ideas: (1) offering software testing as a competitive, easily accessible *Web service*, and (2) doing fully automated testing *in the cloud*, to harness vast, elastic resources toward making automated testing practical for real software.

A software-testing Web service allows users to upload the software of interest, instruct the service what type of testing to perform, click a button, and then obtain a report with the results within minutes or hours. This report is a list of bugs found, or the level of coverage obtained by tests with successful outcomes. Such a service can have a basic interface, where an end user uploads, e.g., the latest Windows service pack and then chooses from a menu of possible test types (e.g., comprehensive testing, security testing). A service can also have an expert interface, to be used by software developers to provide sophisticated definitions of what "a bug" may be, thus teaching the testing service what kinds of correctness violations to look for. For professional uses, TaaS can integrate directly with the development process and test the code as it is written.

We wish to empower consumers of software—both programmers and end users—to be in control of the quality of the software they use. Information on the bugs present in a piece of software is often (partially) known to the vendor, but not to consumers, for both technical and business reasons. With TaaS, we aim to make this otherwise-hidden information openly available on-demand to anyone who wishes to obtain it. Software testing ought to be fast, automated, and as easy and accessible as Web email.

TaaS can also serve as a publicly available certification service, that enables comparing the reliability and safety of software prod-


*SoCC'10,* June 10–11, 2010, Indianapolis, Indiana, USA.

ucts. In this way, TaaS can promote open competition among software vendors and compel them to produce software that is reliable.

Doing automated testing in a cloud instead of on individual developers' machines increases the available compute power by orders of magnitude. In the past, faster CPUs enabled increased levels of interactivity in development, such as quick compile-retry cycles. Cloud-based computation, offering vast numbers of fast CPUs with plenty of memory, could engender a similar transformation, with TaaS becoming a seamless extension of a developer's environment. If automated testing techniques can be adapted to scale up on cloud infrastructures, they can yield the order-of-magnitude lower bug density and higher programmer productivity we seek.

### *Our Goal in Brief*

Automated software testing, available to anyone and everyone at low cost, can transform the current development paradigm into one that involves more-thorough yet less-time-consuming testing. Our end goal is for all software to be more reliable and safe. Toward this goal, we see four fundamental research challenges: testing must (a) be fully automated, i.e., humans no longer write test harnesses; (b) scale to frequently-changing code bases that exceed 1 million lines of code; (c) be feasible as a service, i.e., useful to developers/consumers and economically viable; and (d) be able to directly test binaries, since much software is still proprietary.

Individual software testing continues to be relevant even as SaaS (software as a service) gains increasingly more ground. End users, as well as organizations and corporations, rely on ever more third-party software. Software services are stitched together from large volumes of third-party code (libraries, databases, Web servers, virtual machines, etc.). Consumers install, run, and upgrade software not only on their increasing number of computers, but also on their mobile phones, audio/video players, TiVo, and cameras.

### *An Historical Perspective*

An important turning point in improving the productivity-to-bugs ratio was brought about by the introduction of high level languages and compilers in the 1950s, gradually eliminating most direct use of assembly language. Another important development was faster hardware and compilers, which now provide programmers *quick feedback* on syntax errors and low-level programming errors during the build process. These two events transformed programmers' attitude toward writing code: less concern for the minor details and more time devoted to the higher level thought process.

We expect TaaS to similarly transform the way we write code, by providing prompt feedback on higher level programming errors and enabling developers to spend more time thinking about system-level properties instead of low-level details. TaaS can provide feedback on semantic correctness, instead of mere syntax. Quick feedback on code quality during the development process enables programmers to build systems that are closer to being correct.

The compute power needed to achieve prompt feedback on deep program properties, such as whether a particular *assert()* could ever fail or not, far exceeds what is available in a mere workstation. Cloud infrastructures make such compute power available today, if only we had the automated test techniques to harness it.

In this paper, we make the case for TaaS, hoping to motivate also other researchers to engage in adapting automated testing techniques to the cloud. We first describe in more detail the three variants of TaaS (§2), present our initial forays into cloud-based automated testing, along with ideas for future steps (§3), make the social and business case for TaaS (§4), describe the expected benefits and drawbacks of TaaS (§5), and finally close with a list of research challenges (§5) and conclusions (§6).

## 2. SOFTWARE TESTING WEB SERVICES

While the most obvious embodiment of TaaS is a service aimed at software developers, we see TaaS reaching further: end users themselves could use TaaS, with the same ease with which they use Web email. In this section, we describe a form of TaaS aimed at developers (§2.1), then TaaS for end users (§2.2), and finally TaaS as a universally accessible certification service for software (§2.3).

### 2.1 TaaS$_D$ for Developers ("Sidekick")

TaaS$_D$ can become a programmer's true sidekick, i.e., an inseparable companion assisting the developer at every step.

#### *Continuous Testing*

In the simplest form, a TaaS$_D$ provider operates "in a loop" that pulls the latest code from the developers' repository. It then exercises the various paths through the code and checks them against a collection of so-called test predicates (described in more detail below). Continuous testing integrated into the development environment has been previously proposed [17] as a way to run developer-provided test suites in the background on the developer's workstation. In TaaS$_D$, however, developers provide higher level specifications of what should be tested: instead of imperative test suites, they write test predicates, which takes considerably less human time.

This has two benefits: it places less burden on the developer, and it allows checking much deeper properties faster, by using the resources of the cloud. TaaS$_D$ continuous testing can improve software reliability and shorten the development cycle by automating test generation and finding bugs as software is being developed, even before a test harness exists. This forces developers to produce high quality code early on during development, when bugs are cheapest to eradicate [13]. Only a cloud environment could allow TaaS$_D$ to provide quick feedback, i.e., reduce what would otherwise take several days down to mere minutes or seconds.

#### *Test Predicates*

Predicates over program state or control flow can succinctly characterize undesired behaviors. Test predicates can be, for example, a more sophisticated form of *assert*-like statements. They can use abstract, symbolic program state to specify computation properties; e.g., "if ever $factorial(\lambda) \neq \lambda * factorial(\lambda - 1)$, that is a bug." A testing service smartly exercises as many execution paths through a program as possible and checks whether there exist paths that trigger these test predicates. In our simple example, TaaS$_D$ could find concrete $\lambda$ inputs for which the above predicate is true.

Test predicates fall into two categories: universal predicates and application-specific predicates. Universal predicates are broadly accepted as describing bugs, such as dereferencing a null pointer, entering a deadlock, race conditions, memory safety errors, crashes. Such predicates can be either given as declarative expressions or encoded in write-once/use-many imperative checkers [14]. Application-specific predicates capture semantics that are particular to the tested program (e.g., $numConnections > maxPoolSize + delta$). For every predicate violation, TaaS$_D$ produces a set of inputs, environment conditions, and sequence of program events that developers can use to reproduce the corresponding bug [18].

Application-specific test predicates often come to one's mind while coding. Such properties cannot always be captured in a local *assert()* statement in the code, but rather require a more global predicate over program behavior. Should the scope of a predicate need to be restricted to a portion of the code, it can be done by incorporating a range of line numbers in the predicate itself.

We envision allowing developers to write these test predicates and upload them into a database at the TaaS$_D$ provider. They could

be uploaded manually via a Web interface, or be provided directly from withing the IDE (e.g., Eclipse or Microsoft Visual Studio).

We believe that test predicates, although not suitable for expressing absolutely all bugs, can be used for many classes of bugs. While bugs can relate to arbitrarily complex semantics, many of the bugs that plague today's software are buffer overflows and other memory errors, crashes, integer overflows, race conditions, deadlocks, etc. all of which can be easily encoded in test predicates. For most bugs that violate higher level program semantics, *assert*-style predicates over the global program state are also sufficient. For the remaining types of bugs, there exist more sophisticated forms of expressing them, such as formal logics that capture temporal properties, or, as a last resort, imperative test programs.

TaaS$_D$ provides an entire spectrum of solutions to developers: they can rely solely on the fully automated discovery of bugs that can be detected by universal predicates, or they can provide their own test predicates or imperative test suites. The benefit of TaaS$_D$ is that it uses the resources of the cloud to run this predicate checking on many more execution paths than would be feasible in the developer's own infrastructure. In this sense, TaaS complements advances in programming languages—strong type systems, for example, prevent developers from making low-level mistakes, while TaaS helps check the next-higher level of bugs.

We plan to run TaaS$_D$ as a "public service" for open-source software developers. This effort could make open-source code the most reliable software available. For such a public service, we expect the user and developer community to be willing to contribute detailed test predicates to a Wikipedia-style database. To this end, we are building a system, called Cloud9, which promises to scale symbolic execution [10]—a popular test automation technique—to large clusters of machines. Preliminary results [3] show substantial speedup over a single-node state-of-the-art symbolic execution engine when testing real UNIX utilities. The key techniques underlying Cloud9 are summarized in §3.

## 2.2 TaaS$_H$ for End Users ("Home Edition")

TaaS$_D$ helps develop more reliable software, which makes end users happier. But can TaaS *directly* benefit end users? Yes, it can.

Consider the following scenario: Mrs. X, a grandmother who lives by herself, owns a computer and a mobile phone. She relies on the mobile phone to notify her children (who live in the same city) whenever she experiences the symptoms that often precede her seizures. The software on her mobile phone recently notified her that it needs to be upgraded, to improve the speech recognition component. Mr. X knows that such upgrades are perilous, and that a buggy upgrade may disable her phone altogether. At the same time, improved speech-to-text would help the phone better handle her aging voice. Mrs. X is the kind of user who can benefit from a "home edition" version of TaaS, which we refer to as TaaS$_H$.

The key difference between TaaS$_H$ and TaaS$_D$ is the service interface and the presentation/interpretation of results. Developers can be expected to write test predicates, but end users cannot. Thus, whereas TaaS$_D$ checks for both universal and application-specific test predicates, TaaS$_H$ only checks for universal predicates.

We expect end users to employ testing services in a "one-of" manner, unlike developers who will likely prefer continuous testing. Thus, TaaS$_H$ has a public website where consumers can upload software in binary or bytecode form and select from a pull-down menu the type of testing they want. Using this testing service should be no more complex than downloading software from the Web. By default, TaaS$_H$ may check programs for bugs like buffer overflows, deadlocks, or race conditions, but the TaaS$_H$ provider is free to tap into additional databases of test predicates.

Within minutes, the TaaS$_H$ service produces a webpage with the results of the tests, indicating whether it found any serious bugs, such as hangs or crashes. A bug is automatically rated as serious vs. minor based on the corresponding test predicate, itself rated by the TaaS$_H$ provider or the predicate writer. Mrs. X allows the phone to update itself only if the test report says no serious bugs were found. For interested users, the TaaS$_H$ response may include a rating of the software, akin to stars for products on e-tailers' websites.

This service would cost Mrs. X no more than a few cents, or even be freely included in her monthly phone subscription. Even though the TaaS$_H$ provider commissions a few dozen machines for several minutes to run the test, this cost can be amortized across multiple users: if this same upgrade has already been tested before, the response to Mrs. X can be immediate and cost the provider virtually nothing. As will be seen later, TaaS offers attractive opportunities for economies of scale, especially for widely used software.

Since TaaS$_H$ emphasizes simplicity, it only checks for a set of "canned" bad behaviors, such as memory safety bugs or deadlocks. Community efforts, however, are likely to produce additional test predicates, in the spirit of Wikipedia or Knol [7], perhaps based on bug reports filed in the past. Such a database of test predicates could then be tapped by a TaaS$_H$ provider for the benefit of its users.

The TaaS$_H$ scenario presented here is not far-fetched. We have built a tool, called DDT [11], for testing closed-source binary device drivers against undesired behaviors, like race conditions, memory errors, resource leaks, etc. DDT combines virtualization with a specialized form of symbolic execution to thoroughly exercise tested drivers; a set of modular dynamic checkers use test predicates to identify bug conditions. In preliminary experiments, DDT tested six mature Windows-certified closed-source binary drivers for less than 5 minutes each and found 14 different serious bugs. DDT produces executable traces for every path that leads to a failure, thus proving the existence of the bugs and helping developers debug them. The test predicates used in DDT were extracted from the Microsoft Driver Verifier [14], shipped with Windows, plus a few new ones added by us.

While TaaS$_H$ does not offer much flexibility, it is still a compelling service for end users like Mrs. X, who otherwise would have to blindly trust software vendors.

## 2.3 TaaS$_C$ Certification Services

The third type of TaaS is a public certification service, which provides an objective assessment of a software product's quality. A primitive form of certification is already gaining hold in the industry, as in the case of Microsoft's Hardware Quality Labs testing of third-party software, or Apple's certification process for listing applications in its App Store. TaaS$_C$ analyzes software (either in binary or source code form) and, for each defect found, provides irrefutable evidence of the defect. Based on the defect density, TaaS$_C$ can provide a rating for each product. For an industry to compete on a certain product attribute, that attribute must be easily explained and quantified for consumers. It is for this reason that software companies compete on performance (measurable through benchmarks) and on features (measurable via check lists). We believe this is also the reason for which the software industry has not started yet seriously competing on reliability, safety, and security.

In proposing a software certification service, we draw inspiration from Underwriters Laboratories Inc. (UL), an independent product safety certification organization in the USA. UL has had a universally recognized positive effect on the manufacturing industry, encouraging the adoption of important safety measures.

TaaS$_C$ is meant to be the Underwriters Labs of the software industry. Similar to UL, the certification service tests and then certifies (by digitally signing) the tested software—this is the equivalent of the UL Mark. The TaaS$_C$ provider can be funded by governments, by industry consortia, or simply provide certification services for pay. It can offer different levels of certification, depending on the types of predicates that are checked. When a product is modified, it needs to be re-tested and re-certified. In an ideal future, software companies will be required to subject their software to quality validation on such a service, akin to mandatory crash testing of vehicles. In the absence of such certification, software companies could be held liable for damages resulting from bugs.

An officially sanctioned TaaS$_C$ provider maintains a Web-accessible directory of the software products it tested and certified; this list can be used by consumers to compare products. Certification, of course, does not guarantee the product will perform acceptably or that it is safe under all conditions (such as misuse), but it provides an increased level of assurance. Being certified would not carry legal weight (as does, for instance, the European CE Mark or the FCC Part 15 requirement for electronic devices), but we expect that it would become difficult in practice to sell software that does not carry such a certification. IT consulting firms may be unwilling to install uncertified products, use of uncertified software may invalidate certain insurance coverages, and governmental authorities could require contractors to use exclusively certified software.

In addition to certification, a TaaS$_C$ provider could also publish sorely needed statistics on software: Which bugs are the most prevalent? What is their frequency? What is the typical bug density for each class of applications? Such data would help everyone doing research on reliability (systems, databases, programming languages, etc.), the same way surveys and studies help the medical profession and the pharmaceutical industry. It would enable the development of more scientific and rigorous approaches to software development, grounded in concrete data.

# 3. INFRASTRUCTURE AND SOFTWARE

Automated testing relieves humans from the task of writing test cases and workload drivers. For such a testing technique to be feasible for TaaS, it must be able to control program execution, so that it takes the program through as many different execution paths as possible, and be able to automatically recognize undesired behavior along those paths, i.e., find bugs. Even if a technique cannot determine 100% that a system is bug-free, by exploring substantially more execution paths than what a human-written test could do constitutes a valuable service for developers and end users.

There are several ways to construct such automated testing services. In our work, we use a technique called symbolic execution [10] combined with test predicates. We are working on parallelizing symbolic execution on large, elastic clusters of machines, in order to allow it to scale up to realistically sized programs. Other techniques, such as structured input generation [15], that can run in parallel on shared-nothing clusters, can also be deployed in TaaS.

## *Classic Symbolic Execution*

Symbolic execution is a technique for automated testing, originally proposed in the 1970s. It has recently been shown to find (with no human assistance) bugs that were missed by manual testing and static analysis [4, 6, 12, 1]. Instead of running the program with regular inputs, a symbolic execution engine runs the program with abstract, symbolic inputs that are unconstrained, e.g., an integer input $x$ is given as value a symbol $\lambda$ that can take on any integer value. When the program encounters a branch that depends on $x$, program state is forked to produce two parallel executions, one following the then-branch and another following the else-branch. The symbolic values are constrained in the two execution clones so as to make the branch condition evaluate to true (e.g., $\lambda<0$), respectively false (e.g., $\lambda\geq 0$). Execution recursively splits into sub-executions at each relevant branch, turning an otherwise linear execution into an execution tree that captures all possible executions (Figure 1).

```
void write( int p ){
  if (p < MAX) {
    if (p > 0)
      ...
    else {
      ...
    }
  } else {
    if (p > 3)
      close(p);
    else {
      ...
  ...
```

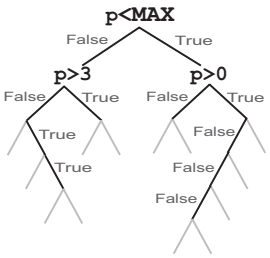


**Figure 1: Symbolic execution tree for an example body of code.**

Symbolic execution consists of systematically exploring this execution tree. Each inner node is a branching decision, and each leaf is a program state that contains its own address space, program counter, and set of constraints on program variables.

When an execution encounters a bug, as defined by a test predicate, the conjunction of constraints collected from the root to the goal leaf is solved to produce concrete program inputs that exercise the path to the bug. In addition to these constraints, program events (such as thread context switches) must be factored in, as illustrated in our execution synthesis system [18]. Herein lies the strength of symbolic execution: it can automatically generate test cases that evidence bugs. Symbolic execution is substantially more efficient than exhaustive input-based testing—it analyzes code behavior for entire classes of inputs/events at a time, without having to try each one out—yet is at least as complete.

Unfortunately, symbolic execution faces two serious challenges: *high memory consumption* and *CPU-intensive constraint solving*, both of which are roughly exponential in program size. Memory consumption results from the large (potentially infinite) symbolic execution tree. CPU consumption results from the fact that, at each branch instruction, the symbolic execution engine must check which of the branches are feasible, given the current constraints. Consequently, on a present-day computer it is only possible to thoroughly test programs with a few thousand lines of code; for larger programs, only the shorter paths can be explored. Thus, symbolic execution is virtually unheard of in the general-purpose software industry because real programs often have millions of lines of code, and executing them symbolically on a single node is not practical.

## *Symbolic Execution in the Cloud*

We are building Cloud9, a *parallel* symbolic execution engine to run on large shared-nothing clusters of computers, thus harnessing their aggregate memory and CPU resources. In this way, we can mitigate the memory and CPU bottleneck of symbolic execution.

Parallelization is a natural way to improve the scalability of symbolic execution, but doing so in a cluster presents significant challenges. Furthermore, in a cloud setting, parallel symbolic execution requires coping with frequent fluctuation in resource quality, availability, and cost, which are not present in regular clusters.

Cloud9 consists of many worker nodes and one or more coordinator nodes. Each worker independently explores a subtree of

the program's execution tree by running one classic symbolic execution engine and a constraint solver on each CPU core. As it explores paths, Cloud9 checks whether any bug predicates are true.

The choice of which branches to pursue first is governed by a so-called search strategy. In the cloud version of symbolic execution, instead of using a single strategy, we simultaneously employ a portfolio of multiple strategies. This allows the exploration to speculate on the promise of certain paths, taking advantage of the fact that speculation requires solely employing a few additional machines. Our preliminary results indicate that diversification of exploration strategies can help find sooner the paths leading to a desired test goal (such as maximizing code coverage, testing the bounds of all string copy operations, etc.) [3]. This is particularly relevant for symbolic execution trees of infinite size.

One key aspect of making Cloud9 scale to large clusters is performing efficient load balancing at infrequent time intervals. The coordinator reasons about which parts of the execution tree ought to be transferred between workers to distribute load evenly. Cloud9 employs a discrete job model that allows workers to self-regulate, thus requiring the intervention of the coordinator relatively rarely. To balance load, workers exchange compact encodings of tree nodes and quickly reconstruct the state of the migrated node, without having to copy potentially hundreds of MB/state across the network.

Another important aspect is the quality of the program state encoding. Since most cloud clusters employ commodity network interconnects, transferring explicit states over the network often turns into a bottleneck (which is one of the reasons why parallel model checkers have a hard time running on clusters without shared memory). Additionally, we are developing techniques for reducing redundancy, handling worker failures, and coping with heterogeneity.

Since symbolic execution is a dynamic testing technique, it has no false positives. This means that bugs found by a TaaS service are legitimate bugs accompanied by inputs and a set of system events that help reproduce them. A recent study [8] showed that lack of false positives and the ability to reproduce bugs not only aids but also compels software developers to fix bugs sooner.

When testing software that depends on hardware features, as is the case of device drivers or a mobile phone operating system, it may appear necessary for the TaaS provider to employ hardware simulators. However, this challenge can be circumvented with symbolic hardware. DDT [11] showed that this approach requires neither real hardware nor hardware models to test device drivers—instead, symbolic hardware returns symbolic values to the software, thus testing it against all possible reactions of the hardware.

### *Infrastructure*

We expect TaaS providers to either operate their own data centers, or provide TaaS as a value-added service on top of a public cloud operated by a third party, such as Amazon EC2.

TaaS in public clouds is an ideal solution for small and medium companies, which cannot afford the upfront investment of setting up large clusters of machines. Moreover, even for large companies that do have their own clusters, TaaS can provide a way to accommodate spikes in their resource needs ("cloud bursting"), such as may be required during intensive test cycles prior to a release. Finally, TaaS can provide an incremental path to gradually move testing from in-house under-provisioned clusters into the cloud.

Besides public clouds, there are two other available variants: private clouds and cooperative clouds. A private cloud may be preferred by large companies that have vast hardware resources they can dedicate to internal use (e.g., Google, Microsoft). A cooperative test cloud is a federation of user machines (similar to SETI@home) or even data centers, pooled together for shared use.

## 4. THE ECONOMICS OF TaaS

We believe TaaS can be operated in a sustainable manner both as a public service and as a business. In this section we argue that there exists a market for automated test services, we suggest a possible pricing scheme, and finally describe how TaaS providers (both public and commercial) can benefit from economies of scale.

Like most Web services, TaaS will first attract the "long tail" of potential users; in this case, it would be those who cannot justify investing in testing infrastructures, or those who wish to cloud-burst during periods of intense testing. Small and medium businesses, as well as open-source software developers, will likely be the first users. TaaS puts at their disposal a testing service on par with (or better than) what the largest software companies have, thus levelling the playing field. Large companies that already own private clouds can run TaaS as an internal service.

Commercial TaaS providers will have to identify good pricing schemes. Ideally, customers pay proportionally with the value they derive from the testing task they submit to the service. This value can be expressed as a function of the number and importance of the bugs found and/or the confidence that the user gets in the code. Confidence can be expressed in terms of code coverage or path coverage, both of which can easily be measured by the testing service. Alternatively, one may wish to pay per bug found or per unit of coverage increase—in these cases, the marginal value increases over time. For example, achieving an extra 1% coverage on code that is already 95% covered is more valuable than if the starting coverage was only 60%. This telescoping marginal value matches well the increased resources (and thus higher cost for the TaaS operator) required to achieve that target.

TaaS users can provide a target level of desired coverage and/or an upper limit on the budget, and the testing service can optimize accordingly. Since the service provider can allocate resources elastically across its customers, the resource demands of each testing task can be optimized globally across all in-progress test jobs.

For price-sensitive customers, auction-style pricing schemes may be advantageous: depending on how much the user is willing to pay, more or fewer resources can be commissioned in the cloud for that user's task. The difference in price may be reflected in the total time required to test a piece of software, taking longer if it uses fewer resources, perhaps even being suspended for some period of time when there is high demand for resources from other higher-paying customers. Such a mechanism is a good match for auction-based clouds, like Amazon's EC2 Spot Instances, where unused cluster nodes can be employed at very low cost. Perhaps cloud operators will be willing to donate resources to a public version of TaaS, during periods of low utilization.

TaaS providers benefit from economies of scale. First, users are likely to end up testing common bodies of code, like popular libraries (e.g, many Java programs will use the same JDK). The TaaS provider can exploit this redundancy by not re-testing already-tested code, and thus saving resources. The more users a service has, the more exploitable redundancy there will be. Moreover, the provider may choose to test popular bodies of code in advance, during periods when its resources are not in high demand. The initial cost of testing can then be amortized over all the customers who will need those results later on.

Finally, TaaS may introduce new business opportunities. For example, rigorous testing can make it feasible to offer software warranties, which translate into liability payments to the software user in case bugs lead to losses. Such warranties could be backed by insurance products from financial institutions, which would offer software developers an insurance policy in exchange for a premium and a requirement that they use TaaS on their code.

## 5. IMPACT AND CHALLENGES

We expect TaaS to have broad impact on end users, developers, and businesses, leading to higher software reliability in general.

TaaS can both compel and help development organizations to compete based on the reliability of their software products. Certification services provide easily accessible means for consumers to compare product reliability, while testing services help developers write better software. The ease of identifying and reproducing bugs will also shorten the interval between detection and final bug fix.

At the same time, testing services empower end users to check the software they use. Well informed and demanding users will further exert pressure on vendors to produce reliable software. Validation of software could become a built-in feature of operating systems, that transparently checks all new software via a TaaS provider.

The fact that competitors and hackers could use TaaS to find weaknesses in a software product as soon as it is released should compel developers to use TaaS before releasing their software.

On the path to TaaS, there are both technical and non-technical challenges. The first and foremost is finding ways to scale automatic testing techniques to hundreds or thousands of machines in loosely coupled clusters. While progress has been made, for example, in parallelizing model checkers for multi-core CPUs [9], the reliance on shared memory makes the techniques difficult to carry over to clusters. Furthermore, cloud environments present substantial heterogeneity and unpredictability of performance.

Finding incremental testing techniques, which reuse existing test results and compose them with tests focused on new or modified code, can enable $\text{TaaS}_D$ to provide quicker feedback to developers. There is also opportunity for techniques that provide progressive refinement of the test results, so that a coarse grained result can be returned immediately, followed by increasingly more precise results, as they become available.

For test predicates to be easy to formulate and maintain, we must design a language that provides a suitable tradeoff between expressiveness and complexity. Formal logics are powerful, but have proven to be inaccessible to most programmers. Asserts stated directly in the programming language are easy, but less powerful. A middle ground between the two will likely prove the most fruitful.

We need metrics for quantifying the level of confidence we get from a test suite. Common coverage metrics, such as line coverage, do not accurately describe how many of the possible execution paths have been tested. At the same time, an absolute number indicating the number of paths tested is not informative either, and path coverage can rarely be expressed as a percentage, since complex programs almost always have an infinite number of possible paths.

We expect much of the testing in TaaS to be done on binaries, rather than source code. Both Cloud9 and DDT can operate on binaries as well as on source. However, software vendors may be reluctant to allow end users to check the quality of proprietary software using TaaS—they might prevent this through code packing or through licensing terms. It is not clear whether TaaS providers would need to have a license for the software they are about to test. Moreover, exhaustively exercising code paths in binaries may be considered illegal reverse engineering by some vendors, although user demand for reliable software may change that perspective.

Finally, providing confidentiality of tested code may also be a challenge. While this is not necessary in the case of binaries, proprietary source code will need to be kept confidential by either running TaaS in a private cluster, or by providing strong guarantees and legal provisions if doing so on a shared cloud. There can also be concerns regarding export restrictions, if sensitive software, such as cryptographic algorithms, ends up being tested by services operating in countries where that code cannot be legally exported to.

## 6. CONCLUSION

In this paper we made the case for TaaS—automated software testing as a cloud-based service. We presented three classes of testing services: $\text{TaaS}_D$ for developers to more thoroughly test their code, $\text{TaaS}_H$ for end users to check the software they install, and $\text{TaaS}_C$ certification services that enable consumers to choose among software products based on the products' measured reliability.

We argued that the combination of recent advances in test automation and the availability of compute clouds can offer unprecedented levels of testing quality. We find TaaS to be compelling from both technical and non-technical points of view. By simultaneously empowering consumers to make educated choices and also enabling developers to build better products, TaaS has the ingredients to indeed help reduce bug density by an order of magnitude.

## 7. REFERENCES


[1] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[2] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communication Security*, 2006.

[3] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. In *Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[4] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conf. on Programming Language Design and Implementation*, 2005.

[5] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. Technical Report MSR-TR-2007-58, Microsoft Research, 2007.

[6] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symp.*, 2008.

[7] Google Knol. http://knol.google.com.

[8] P. J. Guo and D. Engler. Linux kernel developer responses to static analysis bug reports. In *USENIX Annual Technical Conf.*, 2009.

[9] G. J. Holzmann and D. Bosnacki. Multi-core model checking with SPIN. In *Intl. Parallel and Distributed Processing Symp.*, 2007.

[10] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[11] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conf.*, 2010.

[12] R. Majumdar and K. Sen. Hybrid concolic testing. In *Intl. Conf. on Software Engineering*, 2007.

[13] S. McConnell. *Code Complete*. Microsoft Press, 2004.

[14] Microsoft. Driver verifier. http://www.microsoft.com/whdc/DevTools/tools, 2009.

[15] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with Korat. In *Symp. on the Foundations of Software Eng.*, 2007.

[16] Redhat security. http://www.redhat.com/security/updates/classification, 2005.

[17] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Intl. Symp. on Software Reliability Engineering*, 2003.

[18] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *EUROSYS Conf.*, 2010.