Project title: Enforceable Security in the Cloud to Uphold Data Ownership
Project acronym: ESCUDO-CLOUD
Funding scheme: H2020-ICT-2014
Topic: ICT-07-2014
Project duration: January 2015 – December 2017

D3.2

Report on techniques for security testing

Editors: Ahmed Taha (TUD)
Nicolas Coppik (TUD)
Ruben Trapero (TUD)
Neeraj Suri (TUD)
Andrew Byrne (EMC)
Reviewers: Christian Cachin (IBM)
Stefano Paraboschi (UNIBG)

Abstract

This deliverable is part of T3.4 that covers security testing techniques for collaborative multi-user cloud storage systems. This deliverable provides a study of the different security testing techniques, evaluates their applicability to the different parts of the cloud services and analyzes their appropriateness according to the access/resources available for testing. The deliverable also evaluates the requirements and use cases elicited in WP2 and WP1 respectively to assign the most appropriate security testing technique according to factors such as impact on protected assets or resources involved.

Type	Identifier	Dissemination	Date
Deliverable	D3.2	Public	2016.09.30

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644579. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract No 150087. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission or the Swiss Government.

ESCUDO-CLOUD Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	British Telecom	BT	United Kingdom
3.	EMC Corporation	EMC	Ireland
4.	IBM Research GmbH	IBM	Switzerland
5.	SAP SE	SAP	Germany
6.	Technische Universität Darmstadt	TUD	Germany
7.	Università degli Studi di Bergamo	UNIBG	Italy
8.	Wellness Telecom	WT	Spain

Versions

Version	Date	Description
0.1	2016.09.07	Initial Release
0.2	2016.09.28	Second Release
1.0	2016.09.30	Final Release

List of Contributors

This document contains contributions from different ESCUDO-CLOUD partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Ruben Trapero (TUD), Ahmed Taha (TUD), Neeraj Suri (TUD)
Chapter 1: Introduction	Nicolas Coppik (TUD), Ruben Trapero (TUD), Ahmed Taha (TUD), Neeraj Suri (TUD)
Chapter 2: Security Testing Techniques	Nicolas Coppik (TUD), Ahmed Taha (TUD), Ruben Trapero (TUD), Neeraj Suri (TUD)
Chapter 3: Application Level Verification Techniques	Andrew Bryne (EMC)
Chapter 4: Applicability of Security Testing Techniques to Use Cases	Ahmed Taha (TUD), Ruben Trapero (TUD), Neeraj Suri (TUD)
Chapter 5: Conclusion	Ruben Trapero (TUD), Ahmed Taha (TUD), Neeraj Suri (TUD)

Contents

Executive Summary **9**

1 Introduction **11**
1.1 Interactions with other Work Packages . . . 12
1.2 Problem Definition . . . 13
1.2.1 Terminology . . . 13
1.2.2 Security Testing . . . 14
1.3 Methodology . . . 15
1.4 Outline . . . 16

2 Security Testing Techniques **17**
2.1 White Box Approaches . . . 18
2.1.1 Static Analysis . . . 18
2.1.2 Instrumentation-guided Testing . . . 19
2.1.3 Symbolic Execution . . . 20
2.1.4 Applicability . . . 22
2.2 Black Box Approaches . . . 22
2.2.1 Fuzzing . . . 23
2.2.2 Binary Analysis and Instrumentation . . . 26
2.2.3 Applicability . . . 27
2.3 Model-based Security Testing . . . 27
2.3.1 Model-based Security Testing Techniques . . . 29
2.3.2 Practical Concerns . . . 29
2.3.3 Applicability . . . 30
2.4 Penetration Testing . . . 30
2.4.1 White Box Penetration Testing . . . 31
2.4.2 Black Box Penetration Testing . . . 32
2.4.3 Penetration Test Automation . . . 32
2.4.4 Applicability . . . 32
2.5 Vulnerability Assessment . . . 33
2.5.1 Vulnerability Assessment Tools . . . 33
2.5.2 Vulnerability Rating and Prioritization . . . 34
2.5.3 Applicability . . . 35
2.6 Test Parallelization for Security Testing . . . 36
2.6.1 Applicability . . . 40
2.7 Guidelines, Methodologies and Tools . . . 41
2.7.1 OWASP Testing Guide . . . 41

2.7.2 Open Source Security Testing Methodology Manual 41
2.7.3 Developed Supporting Tools 42
2.8 Summary of security testing techniques 43

3 Application Level Verification Techniques 45
3.1 Code Integrity 46
3.1.1 Build Process Integrity 47
3.1.2 Code Signing 48
3.2 Application Integrity 48
3.2.1 Web Application Security 49
3.2.2 File Integrity Monitoring 53
3.3 Trusted Computing 54
3.3.1 Virtual Machine Integrity 54
3.3.2 Container Integrity 56
3.3.3 Trusted Platform Module 57
3.3.4 Trusted Cloud Computing Platform (TCCP) 62
3.3.5 Direct Anonymous Attestation 63
3.3.6 Summary 66

4 Applicability of Security Testing Techniques to Use Cases 67

5 Conclusion 86

Bibliography 87

List of Figures

1.1 Interactions of T3.4/D3.2 with other WPs of ESCUDO-CLOUD 12
1.2 Life cycle for the provisioning of cloud services and the elements required for testing 16

2.1 A schematic overview of instrumentation-guided testing 20
2.2 High level overview of buffer overflow detection with Dowser 21
2.3 Simple test parallelization using separate machines 38
2.4 PAIN test parallelization on a single machine 39
2.5 PAIN test parallelization using multiple machines 40

3.1 TPM Architecture 58
3.2 vTPM implemented using PCIXCC 61
3.3 vTPM implemented using TPM on motherboard 61
3.4 Trusted Cloud Computing Platform 63
3.5 Trust in Computing 64
3.6 Intel SGX 65

4.1 Process to evaluate requirements vs security testing techniques 68

List of Tables

2.1 Summary of white box security testing techniques for cloud-based software systems 22
2.2 Summary of black box security testing approaches and related techniques for the cloud 28
2.3 Summary of model-based security testing techniques 30
2.4 Summary of penetration testing techniques for the Cloud 33
2.5 Summary of vulnerability assessment techniques for the Cloud 35
2.6 Summary of test paralellization techniques 40
2.7 Security testing techniques over the cloud provisioning life cycle 43

3.1 Selection of Relevant ASVS Controls 53
3.2 Virtual Machine Files 55

4.1 Use Case 1 requirements and their direct and indirect assumptions 72
4.2 Use Case 2 requirements and their direct and indirect assumptions 75
4.3 Use Case 3 requirements and their direct and indirect assumptions 82
4.4 Use Case 4 requirements and their direct and indirect assumptions 85

Executive Summary

With the growth of cloud computing solutions, cloud customers are finding increasing opportunities to move their applications and data to the cloud. Correspondingly, an increasing number of security issues also arise that threaten the security of the sensitive personal information of cloud customers. While the number of cloud computing customers and services is increasing, there is still a conspicuous gap with respect to the adoption of cloud solutions in critical contexts where the protection of customers data is a must.

As a result, cloud services need to provide security mechanisms that help guarantee the protection of their customer data. Security-by-design is one of the approaches that allows to embed security from the inception of the service (from the specification of requirements to the design and the implementation). While this provides a good way to ensure the usage of security mechanisms in cloud services, it is still required to check that these security mechanisms actually work as planned. Hence, security testing becomes a necessity in the service developer community and, of course, in the cloud service provisioning community too. In the case of cloud computing, and more specifically in the case of multi-cloud, it is even more challenging since the access/resources available to check the security mechanisms of cloud providers are limited. Sometimes multi-cloud services are composed of several CSPs (Cloud Service Providers) that belong to different companies and implement different technologies, thus the security mechanisms implemented are different as well. Most of the time it is not possible to actually access the CSPs to check the security mechanisms implemented as they do not permit access to binaries (and of course also not to the code). Sometimes only the interfaces are available though many CSPs do not even allow access to their interface and we only have access at the application level (just as any other customer). As a result, different CSPs would require different security testing techniques that also get applied at different levels of granularity depending on the type of access/resources available (i.e., code, interfaces, or just access to the application as normal users).

This deliverable compiles (along with presenting some developed testing support tools) the advocated security testing techniques applicable to the ESCUDO-CLOUD environment. We have grouped and compared the testing techniques considering varied parameters that allows us to determine in what aspect of the service they should be applied. We have analyzed the cloud service life cycle and the resources needed to use those security testing techniques. We have also evaluated how demanding these techniques are in terms of the need of resources to apply them. For example, techniques that require the code versus techniques that require just interfaces or techniques that can be used just at the application level. We have evaluated these techniques to analyze their applicability to the use cases identified in the WP1 of ESCUDO-CLOUD. More specifically we have elicited the requirements reported in WP2/W2.3 to evaluate aspects such as the resources linked to every requirement (i.e., access control platform, database, ...), the security aspects involved (i.e., confidentiality) or the impact associated to the unfulfillment of that requirement.

1. Introduction

Collaborative multi-user cloud storage systems hold enormous potential for users, for access to cloud-based data processing services and also to store sensitive and personal data. Naturally, this also makes cloud storage become desirable targets for attackers. Consequently, protecting user data stored in such systems and upholding the principles of data ownership are of high importance, especially since (a) consumer-oriented cloud storage services do not have a particularly positive track record with regard to security [MSL+11, Laa11], and (b) the more privacy-oriented services typically lack the collaboration functionality that is characteristic of collaborative multi-user cloud storage systems. This, alongside the concerns about the relinquishment of control and data ownership to a third party, have hampered the adoption of collaborative multi-user cloud storage systems.

Alleviating such concerns is typically addressed by taking measures to avoid vulnerabilities that threaten user privacy or data security throughout the development lifecycle, and also by combining security by-design supported by continuous security testing and verification. To this end, testing techniques are required that are capable of discovering vulnerabilities across different abstraction layers and in a wide variety of different security testing scenarios, including both black and white box testing scenarios. In particular, these security testing techniques need to fulfill the following requirements:

- They need to be suitable for assessing the security of the mechanisms facilitating secure multi-user interactions and sharing, as developed in Task 3.2, and collaborative data processing and queries, as developed in Task 3.3.
- They must be capable of detecting threats to the principles of data ownership. To this end, the newly developed cryptographic layers are particularly important targets for security testing - vulnerabilities in these layers threaten both overall system security and data ownership in particular and vulnerabilities in key management can allow attackers to effectively circumvent most other security mechanisms. Furthermore, securely implementing cryptographic functionality is known to be highly challenging and new vulnerabilities are routinely discovered even in relatively mature implementations of cryptographic protocols.
- They must also consider the other interfaces exposed by collaborative multi-user cloud storage systems. In scenarios where such functionality is only implemented server-side, possibly by third-party service providers, thorough security testing is challenging if not impossible and we are severely limited in terms of the applicable security testing approaches. However, we also consider scenarios where the desired security, privacy and data ownership guarantees necessitate moving such functionality to the user side, making the problem accessible to a wider array of security testing techniques.
- They should minimize the amount of manual effort required for effective security testing in order to accelerate the adoption of these techniques and allow their application to large scale

systems. To this end, efficient test generation mechanisms and automated reasoning techniques that are capable of detecting potential threats with minimal manual interaction by testers are essential. Since model-based approaches to this problem tend to suffer from scalability issues, heuristic and probabilistic approaches, possibly exploiting domain-specific information, are a more effective for large scale software systems. While such approaches do not offer theoretical completeness, they have the advantage in terms of practical efficiency.

Over the course of this document, we outline various approaches to security testing and discuss their applicability to the different ESCUDO-CLOUD use cases. For context, Section 1.1 overviews the interactions between this document and other ESCUDO-CLOUD work packages. The problem statement and the used terminology are provided in Sections 1.2 and 1.2.1, respectively. The chapter concludes with an outline of the structure of the rest of this document in Section 1.4.

1.1 Interactions with other Work Packages

The work discussed in this document is related to and interacts with the other ESCUDO-CLOUD work packages as depicted in Figure 1.1.

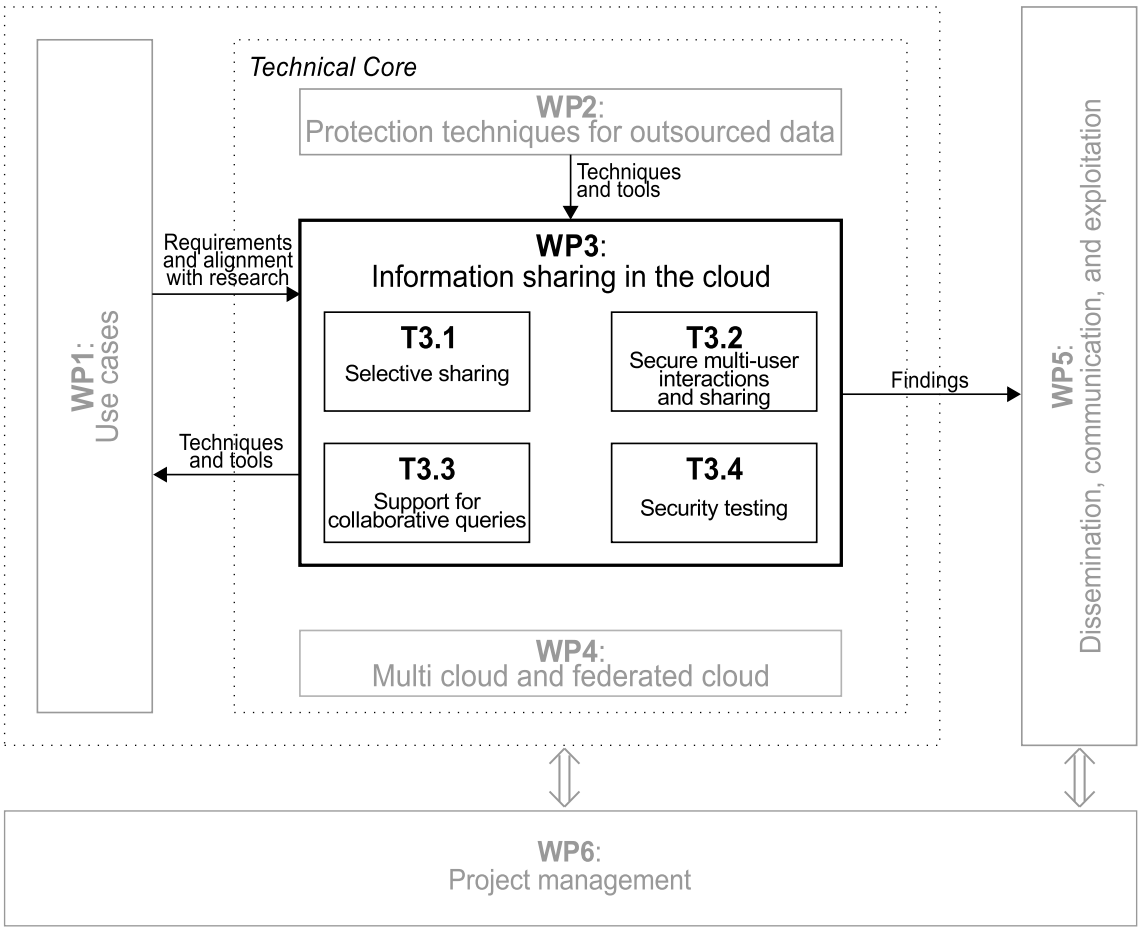

Figure 1.1: Interactions of T3.4/D3.2 with other WPs of ESCUDO-CLOUD

- **Work Package 1** provides the validation use cases for ESCUDO-CLOUD. The use cases and related requirements from deliverables D1.1 and D1.2 form the basis of the security

needs that the techniques provided in this document are intended to address, and we discuss the applicability of these techniques to the use cases. In turn, the techniques discussed in this report will be applied to the use cases as part of WP1 and the results will be communicated in D1.3, D1.4 and W1.2.

- **Work Package 2** provides the design and implementation work of protection techniques for outsourced data. The techniques and tools provided by WP2 are considered as part of the applicability discussion in this document and as potential targets for security testing.

- **Work Package 5** receives the security testing techniques discussed in this document and their applicability to the use cases provided by WP1 for the purposes of dissemination and exploitation.

1.2 Problem Definition

Over the course of this section, we provide an overview of the problem we are tackling with the techniques presented in the following chapters, starting with a discussion of terminology in Section 1.2.1. We then discuss the problem of security testing in the larger context of general software testing in Section 1.2.2 followed by discussions of the two subcategories of functional and non-functional security testing and an overview of software bugs that typically lead to vulnerabilities.

1.2.1 Terminology

Security is typically defined in terms of the classic *CIA Triad* consisting of *Availability*, *Confidentiality* and *Integrity*. These three key terms are defined as follows [ALRL04]:

- *Availability* refers to the readiness of a system to provide correct service to authorized users.

- *Confidentiality* refers to the unauthorized disclosure of information, whether to insufficiently privileged users or to third parties altogether.

- *Integrity* refers to the absence of improper or unauthorized alterations of the system.

Examples of attacks that allow an attacker to impair the availability of a target system are not just straightforward denial of service attacks but also vulnerabilities through which, for instance, an attacker-submitted value may cause the target system to crash. Violations of confidentiality, on the other hand, allow an attacker to gain access to data without possessing the required authorization. A common example are SQL injection attacks wherein attackers may gain access to database contents that should otherwise be unavailable to them. SQL injection attacks may furthermore result in violations of integrity if they also allow the attacker to modify rather than just obtain database contents.

Another important security property is *non-repudiation*, which can be regarded as either a part of integrity or separately on its own. In the context of security testing, the former makes more sense: Systems that require non-repudiation as a security property typically rely on cryptographic mechanisms to assure it. Vulnerabilities that would allow an attacker to bypass these mechanisms would also violate the integrity of the system. Consequently, in this document, non-repudiation is not treated as a separate property from integrity.

1.2.2 Security Testing

In general, software testing is an essential part of the software development lifecycle, indispensable for detecting and removing faults as early as possible. Since it is impossible to test all conceivable preconditions, all conceivable inputs and all conceivable interactions, software testing is inherently incomplete. Consequently, significant effort has been devoted to finding approaches that are capable of prioritizing testcases and making software testing as efficient as possible. Most such approaches deal primarily with the generation or prioritization of unit or component tests, both of which generally fall under the category of *functional testing*. Most security testing, on the other hand, is nonfunctional, more closely related to stability or reliability testing [MR05], which poses its own unique challenges. We conclude this section by discussing how software bugs commonly cause vulnerabilities and which kinds of bugs are particularly closely linked to security issues.

Functional Security Testing

Functional security testing is the application of functional testing mechanisms to security-related functionality, such as, for instance, the functional testing of an authentication mechanism. This area is generally well-covered by established techniques for functional software testing. Since functional security testing only covers a relatively small part of the wider problem of security testing and offers few additional challenges compared to general functional software testing, it is not discussed in further detail in this document.

In general, most such testing deals with what is known as *positive requirements* - requirements that are typically of a form similar to "if presented with a certain input, the program should behave in a certain way or produce a certain output". Such requirements work well for the testing of some security mechanisms, for instance a requirement to enforce increasing waiting times following login attempts with incorrect passwords or a requirement to disable a user account following a certain number of failed login attempts. Such tests can easily be integrated into the normal, non-security related testing lifecycle, although particular care should be taken when specifying test cases for security-related functionality. Other properties that a software system must possess in order to ensure its security, however, can not be specified as positive requirements, such as the absence of memory corruption vulnerabilities. Testing for such properties is the bulk makes up the bulk of security testing and is discussed in the remainder of this document.

An overview of various functional testing techniques and approaches and their applicability to functional security testing can be found in [MR05].

Nonfunctional Security Testing

The bulk of work in the security testing area deals primarily with nonfunctional security testing, that is, it deals with attempting to find defects in software systems that can allow an attacker to violate security properties, even if such faults do not affect the functional correctness of the system. Understanding the challenges associated with security testing first necessitates understanding what kind of defects, faults and shortcomings are relevant for nonfunctional security testing.

Security Bugs

A wide variety of defects can possibly affect system security, ranging from straightforward vulnerabilities such as buffer overflows or other memory corruption vulnerabilities to more subtle faults

that can, for instance, expose sensitive information to an attacker through minor differences in execution timing. Thus, it is not always straightforward to determine which defects have an impact on system security or how large that impact is. Furthermore, systems may be vulnerable to attackers due to a combination of several defects, none of which would result in a vulnerability on its own [AAD+09]. In order to cope with these challenges, many security testing approaches focus on specific classes of bugs that are known to cause vulnerabilities with a very high probability, such as buffer overflows, or more generally memory corruption bugs [SPWS13]. This class of bugs includes all bugs that allow an attacker to trigger a memory error, typically by making a pointer go out of bounds or by causing it to be used after the memory it pointed to has been deallocated. While numerous mitigation techniques at different stages of exploitation exist for such bugs, they are nonetheless still common in low-level languages that do not enforce memory safety, such as C or C++. Furthermore, since C and C++ are still frequently the languages of choice for library development, even applications that are themselves written in languages that do provide memory safety may be vulnerable to such bugs, depending on the libraries they use. Well-known recent examples of vulnerabilities resulting from lack of memory safety are the Heartbleed vulnerability [CVE14] or [CVE16], a remote code execution vulnerability in numerous Symantec products [CVE16].

Such vulnerabilities resulting from violations of memory safety, alongside input validation errors, are the subject of numerous proposed security testing techniques, several of which we discuss in this report. They can be detected on either the binary or source code level and are generally detectable without domain- or application-specific knowledge.

Other approaches focus on finding vulnerabilities at a higher abstraction level, such as application-level vulnerabilities that are frequently caused by mismatches between the specification and the implementation of an application. Examples of this are vulnerabilities wherein certain access control mechanisms are not implemented as specified, allowing users to access data or functionality they should not be able to. Since formal specifications of the expected behavior of such applications are rarely available, security testing for application-level vulnerabilities makes extensive use of heuristics and domain-specific assumptions to determine what would constitute undesirable behavior and strongly favors efficiency over completeness.

1.3 Methodology

Testing techniques vary in utility and applicability depending on different aspects such as the availability of artifacts including, among others, binaries, source code or level of detail of interface/operational specifications. As a result we first need to classify the potential techniques depending on their applicability. A natural reference to use is the life cycle of the cloud service and subsequently assessing the type of tests suitable for each stage of the life cycle. Typically, 5 steps can be associated to the cloud service provisioning as:

- **Specification**. In this stage the requirements of the cloud service are defined. Testing techniques should be able to trace that all the defined requirements are implemented and fulfilled.
- **Design**. In this stage the software components that are part of the cloud service and their interfaces are identified.
- **Coding**. In this stage the components identified in the design are implemented and integrated.

- **Verification**. In this stage the components are integrated and operational. The tests should be able to check if the system is working according to the requirements defined in the specification stage.

- **Operation and Maintenance**. In this stage the service is running and in use by the customers. The testing techniques should be able to check that the service is conforming to specifications and free of vulnerabilities that might affect the specified operations. For some testing techniques, information about customers' interaction with the cloud service and execution logs and traces may also be used.

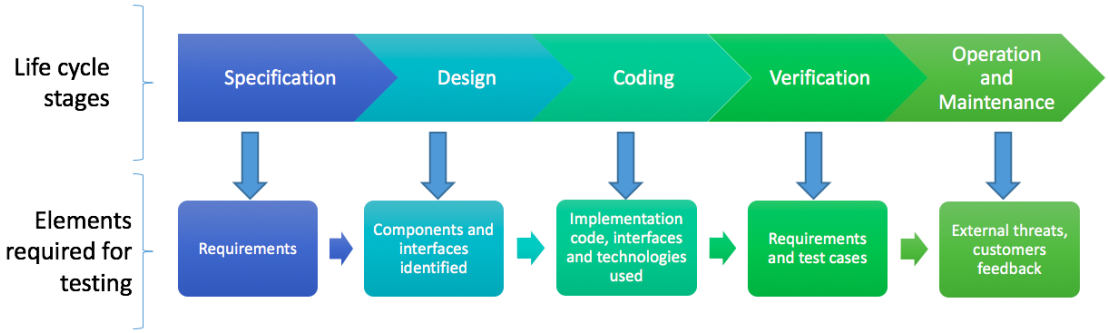

Figure 1.2: Life cycle for the provisioning of cloud services and the elements required for testing

The definition of the security testing techniques (Chapter 2) is developed according to the above described classification. This allows to identify the suitable security testing techniques for each stage of the life cycle in order to further evaluate the system requirements (defined in WP2) for every use case (defined in WP1) and the security testing techniques to apply for each requirement.

1.4 Outline

The remainder of this document is structured as follows. Chapter 2 provides a classification and overview of existing security testing techniques, guidelines and methodologies. Chapter 3 outlines some of the available techniques for ensuring the integrity of applications over their development and use. We discuss the applicability of these techniques to the various use cases in Chapter 4. Finally, Chapter 5 presents some summary observations.

2. Security Testing Techniques

The deployment of security and privacy mechanisms is essential in scenarios and applications involving the management of personal data. It is even more important for scenarios in which third parties might have access to personal data, such as SaaS deployments utilizing third party infrastructure or multi-cloud scenarios involving multiple third parties. To this end, it is paramount to use techniques that aim to protect the fulfillment of the security and privacy requirements expected by users. Techniques such as security-by-design are a good starting point to ensure that security and privacy mechanisms are integrated in cloud services at design and implementation time. However, the current changing context of threats and growing vulnerabilities forces to design techniques to verify the expected security and privacy levels. This is typically done in two possible ways:

- Component level techniques, which are capable of operating on individual software components irrespective of whether or not these components directly expose outside interfaces and are therefore applicable to internal components of the cloud service.
- Application level techniques which externally verify the cloud service, either using interfaces or application interaction.

Cloud services, and especially those cloud services that combine more than one cloud (multi-cloud), are highly modular. Therefore, security testing techniques that are focused on common interfaces and work at the application level are desirable in such scenarios. However, as will be described later in this section, there are component level software security testing techniques which are equally applicable to certain parts of multi-cloud services. While certain techniques are mainly applicable to elements that are under the control of the entity performing the security testing (i.e., white box testing as referred in Section 2.1), other techniques need to be capable of operating with limited information since they have to be applied to elements that are not directly available or only available in binary form (such as elements provided by third parties or user-end components). This Task 3.4 studies the support of security testing techniques applicable to interfaces offered by collaborative multi-user cloud storage systems. To this end, special attention is paid to the security testing techniques applicable to user-end components.

In this section, we discuss numerous different techniques that have been proposed for security testing, starting with so-called *White Box* approaches in Section 2.1. These approaches are characterized by their requirement for source code access and can enable thorough and efficient security testing. In scenarios where such source code access is not available, security testers may need to fall back to what are known as *Black Box* approaches, discussed in Section 2.2. While these approaches do not require source code access and are therefore more widely applicable, they generally do not perform quite as well. Next, model-based security testing approaches are briefly discussed in Section 2.3. Penetration testing is covered in Section 2.4, followed by a discussion of

various vulnerability assessment approaches and tools in Section 2.5. Throughout this entire section, we discuss efficiency concerns as they relate to different security testing techniques. Next, we adress how to overcome these issues: We discuss the applicability of test parallelization to security testing and some of the issues that must be considered in this context in Section 2.6 based on experiences with test parallelization in dependability assessment. We conclude this section by giving a brief overview of widely used security testing guidelines and methodologies in Section 2.7.

2.1 White Box Approaches

White box testing describes all testing approaches that utilize knowledge of the internal workings of the component that is being tested. In practice, this typically means source code access is required. For the purposes of security testing, this can result in increased efficiency and allow for the application of static analysis and symbolic execution to guide test case generation in order to maximize coverage and discover a wide variety of vulnerabilities.

Over the course of this section, we first briefly discuss the benefits of static analysis techniques for finding potential vulnerabilities early and cheaply before moving on to guided security testing techniques that utilize approaches such as symbolic execution to leverage knowledge of program internals for better test case generation. We also briefly address the efficiency concerns that go along with the use of such approaches.

2.1.1 Static Analysis

Static analysis techniques operating on the source code level allow for cheap and easy detection of problems that could result in vulnerabilities like those discussed in Section 1.2.2, such as possible buffer overflows, usage of uninitialized variables or insecure usage of certain APIs. Consequently, tools implementing such techniques exist for a wide variety of languages and have achieved significant adoption by software developers. Widely known examples include clang-analyzer[1] for C, C++ and Objective-C and FindBugs[2] for Java. While these static checkers are primarily used to warn developers about code that is likely to be buggy or considered bad practice, they also contain checkers for certain kinds of security issues, such as the aforementioned insecure API usage (for instance, using `gets` in C code or insecure usage of SQL-related APIs in Java that can lead to SQL injection vulnerabilities). Security-specific checkers[3] have also been developed but are not as widely used. Additionally, commercial checking systems and static security testing tools are also available, offering a variety of proprietary analyses of varying complexity. Due to the low manual effort required and the potential to discover potential vulnerabilities early during development when they are typically relatively straightfoward to fix, the use of static analysis tools throughout development is a valuable part of building secure software.

A brief example snippet of C code can be seen in Listing 2.1. The code in question shows a function that uses the value of an integer parameter as part of a call to `malloc`. If `foo` fails to perform the appropriate bounds checking, the multiplication may overflow, in which case malloc would return a significantly smaller buffer than expected. This, in turn, may lead to out of bounds memory accesses with the corresponding security implications. Potential problems like these are trivial to detect using static analysis tools (clang-analyzer is capable of detecting the example

[1] `http://clang-analyzer.llvm.org`
[2] `http://findbugs.sourceforge.net/`
[3] `https://find-sec-bugs.github.io/`

from Listing 2.1) during development, allowing developers to fix such issues before they become exploitable vulnerabilities.

Listing 2.1: C code containing potentially unsafe use of `malloc`.

```
void foo(int buflen){
    ...
    int *p = malloc(buflen * sizeof(int));
    ...
}
```

Modern static checkers are powerful but also enormously complex pieces of software and as such, usually tightly coupled to just one or a handful of programming languages. Recent research suggests that some of this complexity may be superfluous and proposes a simpler, significantly more portable approach to writing static checking systems which can be easily ported to different programming languages or extended to support new checks [BNE16]. Another recent approach suggests tackling the challenges associated with complex checking systems by leveraging machine learning techniques to automatically generate models of vulnerability types which can then be used as static analysis tools [MNC16]. Nonetheless, many of the underlying checks can themselves be fairly complex to develop and execute, and efficiency remains a concern as applying complex static analyses to large code bases may take a prohibitively long time.

Beyond the source code based tools discussed above, other forms of static analysis operate on higher levels of abstraction, sometimes in combination with dynamic analyses, as in Waler [FCKV10]. Such tools target what is known as application level or logic vulnerabilities, a category that is discussed in more detail later on.

2.1.2 Instrumentation-guided Testing

Instrumentation-guided testing, in the context of security testing usually instrumentation-guided fuzzing, described a set of approaches that stem from attempts to improve the efficiency of black box random testing (or fuzzing, discussed in Section 2.2) without utilizing techniques like static analysis or symbolic execution, which tend to suffer from performance and scalability problems when applied to entire large-scale software systems rather than individual components or compilation units. To this end, such approaches typically perform light weight compile time instrumentation of the target program, use that instrumentation to gather coverage information at run time and then use that information for test case generation or prioritization. A schematic overview of the structure of instrumentation-guided security testing techniques can be seen in Figure 2.1. Note that, while the schematic overview presents the generation of an instrumented version of the source code or an intermediate representation and the subsequent compilation to an instrumented binary as two separate steps, in practice these steps are typically closely linked as instrumentation is commonly implemented as a compiler pass.

The most well known tool in this area is AFL [Zal], essentially a fuzzing tool and discussed in more detail in Section 2.2. This and related or derived tools essentially constitute the state of the art in random testing for security. As they are generally capable of operating on binaries (using mechanisms discussed in Section 2.2), albeit possibly with reduced performance or efficiency, when source code is not available, we consider them to be grey box rather than strictly white box testing techniques and do not discuss them in greater detail here.

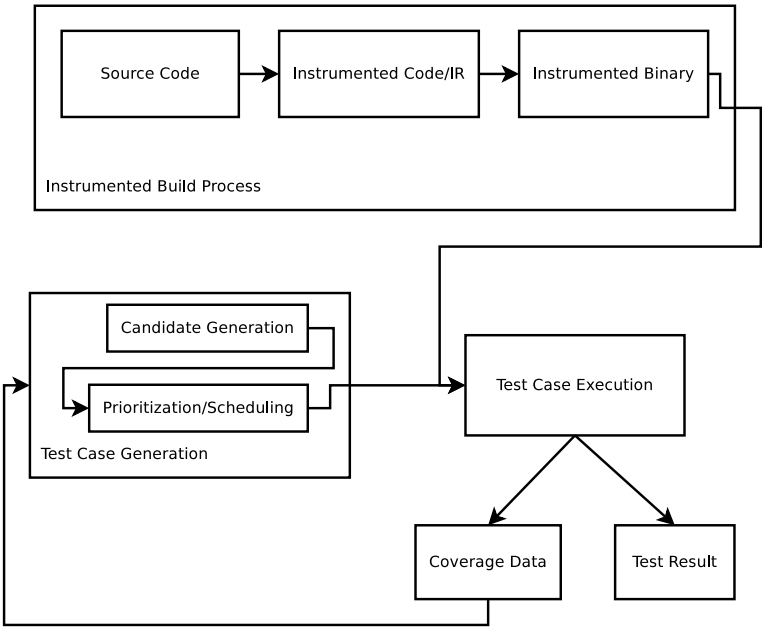

Figure 2.1: A schematic overview of instrumentation-guided testing

The focus on performance that has driven the development of some of these tools and techniques stems from the recognition that security testing in general requires significant amounts of time and computational resources and is, due to this cost, not as frequently applied as it ought to be in practice. As modern systems increasingly rely on parallelization to achieve better performance, security testing must also be parallelized efficiently to remain applicable, but such parallelization is not always straightforward and testers may encounter pitfalls resulting from test interference affecting result accuracy [WSN+15]. This is particularly relevant for those security testing approaches that need to perform hang detection, and we discuss this problem in greater detail in Section 2.6.

2.1.3 Symbolic Execution

Symbolic execution is a program analysis technique in which the target program is executed by an interpreter which treats inputs as symbolic values (in contrast to the concrete values obtained during an actual execution). The symbolic execution engine gathers the constraints on the symbolic values on each path of the program, determining which inputs would trigger the execution of that path. Due to the path explosion problem, symbolic execution generally does not scale well to large software systems, but numerous approaches have been proposed to tackle this problem, for instance by limiting the usage of symbolic execution to certain components or using heuristics to choose certain paths.

In the context of security testing, symbolic execution has been combined with fuzzing (usually a form of black or grey box random testing, discussed in more detail in Section 2.2) in an approach known as white box fuzzing [GLM08]. In white box fuzzing, the target program is first executed non-symbolically with a valid input. The trace from that execution is used to gather constraints on the program input. New inputs are generated by negating individual constraints, and the process is repeated. While this approach is capable of quickly discovering numerous new paths, it suffers from a similar problem as black box fuzzing when dealing with applications with highly structured

inputs, such as compilers: The technique discovers many paths through the early stages of input parsing but is not capable of reaching deeper into the program. As in black box fuzzing, one solution to this problem relies on the use of input grammars, as suggested in [GKL08]: Constraints can be formulated in terms of symbolic input grammar tokens rather than individual bytes, allowing the application of white box fuzzing to complex applications with highly structured input formats.

Security testing techniques based on symbolic execution have also been proposed for specifically targeting certain kinds of vulnerabilities, such as buffer or integer overflows.

For buffer overflows, Dowser [HSNB13] is a guided fuzzer that utilizes symbolic execution and taint tracking (discussed in Section 2.2) in order to discover buffer overflow or underflow vulnerabilities. To this end, the analysis specifically targets locations in the code where array accesses are performed inside a loop, uses taint tracking to determine which parts of the input influence the array indices and then uses symbolic execution to check if there are any values these parts of the input can take that cause a buffer over- or underflow. Figure 2.2[4] illustrates this process for a small example function `foo`, which contains three array accesses inside loops (with `x`, `y` and `z` being loop variables). The approach first detects these accesses using static analysis and then, in the second step, ranks them based on their estimated complexity. Then, in the third step, dynamic taint analysis (see Section 2.2.2) is utilized to determine which parts (i.e., which bytes) of which input affect the pointer that is used in the chosen array access. The identified input parts are then treated as symbolic in the fourth step, and symbolic execution is utilized to exercise the loop containing the chosen array access. Finally, a buffer overflow detection mechanism is used in the final step to detect any buffer overflow that may have been triggered. Dowser is an example of an approach that combines several different techniques relevant to security testing, including symbolic execution and dynamic taint analysis, and attempts to limit the application of symbolic execution to a small part of the input variables and the target program in order to mitigate the efficiency concerns that limit the application of symbolic execution to large scale software systems. These characteristics are shared by many security testing techniques that utilize symbolic execution as testing efficiency is a key concern in this area.

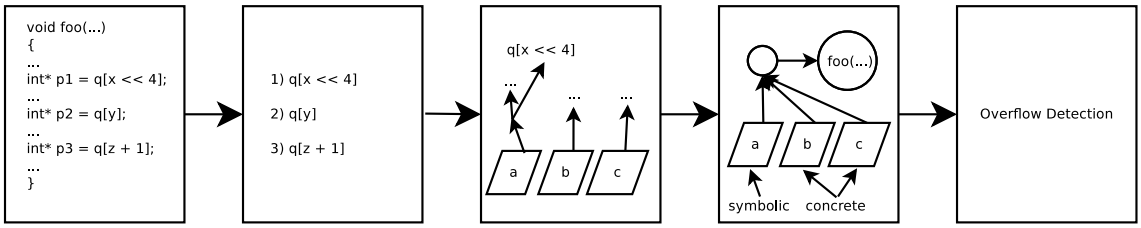

Figure 2.2: High level overview of buffer overflow detection with Dowser

For integer overflow vulnerabilities, DIODE [SDLR+15] identifies target locations and values (such as a call to `malloc` and the corresponding size argument) and uses symbolic execution to determine a symbolic expression for the target value as a function of the program input. This is then used to determine whether there are inputs that both execute the target location and trigger an over- or underflow in the target value. Here, the indentification of specific target locations and values prior to symbolic execution is used to limit the amount of computational effort required to perform the symbolic execution.

The approaches discussed above differ in that some require full source code access while oth-

[4] Illustration based on figure in [HSNB13].

ers are capable of operating on binaries. In general, much of the recent work on symbolic execution builds on KLEE [CDE08], a symbolic execution engine that operates on LLVM intermediate representation. Since lifting binaries to LLVM IR is challenging, such tools generally work best if they have full source code access.

Efficiency and scalability remain concerns with security testing approaches based on symbolic execution. The path explosion problem frequently renders such approaches impractical for larger applications, and while tools that specifically target certain vulnerability classes can improve efficiency, they clearly do not represent a general solution for security testing since a large number of them would be needed to cover even the most common vulnerability types.

Another approach for tackling the path explosion problem is the highly selective application of symbolic execution only as a fallback under specific circumstances. Such a technique has been proposed as part of Driller [SGS+16], which selectively uses symbolic execution to generate new inputs when it encounters conditions that the fuzzing engine is incapable of satisfying. As symbolic execution does not scale well to large software systems, approaches based on selective application of the technique are highly beneficial in testing such systems.

2.1.4 Applicability

Table 2.1 summarizes the characteristics of the security testing techniques described in this section and their applicability in the context of modern cloud-based software systems.

Table 2.1: Summary of white box security testing techniques for cloud-based software systems

	Stage of Life Cycle	**Artifacts required**	**Level of practical adoption**	**Available tools**
Static analysis	Coding or later	Source code or binaries (binary/grey box approaches reduce efficiency)	High	clang-analyzer, FindBugs, may others including several security-specific checkers available.
Instrumentation guided testing	Coding or later	Source code or binaries (binary/grey box approaches reduce efficiency)	Medium	AFL, numerous fuzzers.
Symbolic execution	Coding or later	Source code (some binary tools exist)	Low	Driller, several KLEE-based tools.

2.2 Black Box Approaches

Black box security testing techniques are all those approaches which do not depend on knowledge of the internal structure or workings of the target system or component. Instead, such security testing focuses on interfaces between the target component or system and other components or systems. The most well known black box security testing technique is classical black box fuzzing, the discussion of which makes up the bulk of this section, including a brief overview of some widely used fuzzing tools. We also discuss approaches to move from black box to grey box testing by recovering partial information about a target component or systems internal workings and structure from binary code in situations where source code access is not available to testers.

2.2.1 Fuzzing

Fuzzing is a form of black box random testing. The term was first used by Miller et al. in 1990 [MFS90] in reference to an approach for assessing the reliability of UNIX utility programs. Despite the technique's origins as a dependability testing approach, fuzzing has proven to be an immensely useful and powerful tool for security testing and plays a key role in the large scale, automated security testing of widely deployed software today. We start this section with a brief description of fuzzing in general and an overview of the different categories fuzzing-based approaches to security testing can be classified into before discussing the categories of mutation-based and generation-based fuzzing in more detail. For each category, we give an overview of the category and highlight techniques and recent developments that are particulary relevant to cloud-based software systems in general and the ESCUDO-CLOUD project in particular.

As mentioned above, although similar techniques were known previously, the term fuzzing or fuzz testing itself was first used by Miller et al. [MFS90] for a technique for finding reliability issues with UNIX utility programs by feeding them random input data. Their experiments were originally motivated by the observation that corrupted inputs resulting from line noise would frequently cause UNIX utility programs to crash when working remotely. Fuzzing, then, is originally a technique for finding dependability issues, not strictly security testing. Its relevance to security testing stems from the fact that a bug that causes a program to crash on random input must be considered a vulnerability affecting availability when that input is untrusted. Furthermore, the kinds of bugs underlying such crashes are frequently memory safety violations and therefore likely to be exploitable, a fact that was already noted in [MFS90], which listed the use of fuzzing for security testing as potential future work.

Research on fuzz testing generally deals with finding effective strategies for input data generation, test case prioritization and the application of fuzzing to avenues of program "input" that were not previously considered, such as configuration files (e.g. [DMK10]).

For input data generation, fuzz testing approaches can be broadly classified as either generation-based or mutation-based (or mutational) [MP07]. As the names imply, the former group of approaches *generates* input data from scratch, typically based on some knowledge about the structure or format of the input of the target program, whereas the latter approach works by corrupting or *mutating* known valid inputs according to some strategy, usually either random, heuristic or instrumentation-guided. The advantages and drawbacks of either category are discussed in more detail in the following and relevant examples of each are provided. We conclude this chapter with a brief overview of fuzzing-based security testing in practice and discuss some relevant projects.

Generation-Based Fuzzing

Generation-based fuzz testing approaches generate input data from scratch based on knowledge about the format or structure of the input that the target program expects. The reasoning behind this approach is based on the observation that fuzzing with purely random input data suffers from significant efficiency limitations, particulary for programs that operate on highly structured input, such as programming language interpreters or image processing libraries. In such programs, purely random input is usually quickly discarded by the parsing stage as invalid and the code that actually processes valid or nearly valid inputs is never executed, making it impossible to detect security vulnerabilities in large parts of the target program. Ideally, generation-based fuzzing can avoid these issues and exploit input format knowledge to cover a larger fraction of the target program than purely random or mutation-based fuzzing (which relies heavily on the quality of the

initial seed inputs, as discussed later on).

Exploiting knowledge about the input data format or structure naturally requires that knowledge to be incorporated in the fuzzing tool in some manner. This can be either done manually, as part of the creation of a generation-based fuzzer, or automatically based on a formal specification of the input format if such is available. This represents additional effort that is not required in purely random or mutational fuzz testing.

There are a number of generation-based fuzzers and fuzzing frameworks available that facilitate the generation of fuzzing inputs based on specific rules, templates or grammars, such as the SPIKE fuzzer creation kit [Ait02]. Examples of generation-based fuzzers that are tied to specific input formats are *jsfunfuzz* [Rud07], a fuzzer that was specifically designed for testing the JavaScript engine in Mozilla Firefox, and *CSmith* [YCER11], a tool capable of generating random but standard-conforming C programs for compiler testing. Interpreters and compilers are a common target for generation-based fuzzing due to their typically well-specified input format and their high complexity, which renders them particularly bug-prone.

A further refinement to the grammar-based input generation technique has been proposed in [HHZ12]. Here, instead of using a grammar specification to generate program inputs, the authors propose using the grammar to derive a parser, identifying *fragments* (effectively examples for non-terminals in the input grammar) in a set of sample inputs and using this semantic knowledge to more effectively mutate inputs to generate new test cases. Just as much of the recent work on security testing is moving away from a strict separation of black and white box techniques towards grey box techniques, this example illustrates that research on fuzzing is moving towards finding effective ways to combine generation-based and mutation-based techniques in order to maximize fuzzing efficiency and usability.

Mutation-Based Fuzzing

In contrast to the generation-based fuzzing techniques discussed above, mutation-based fuzzing does not create inputs from scratch or require any protocol- or application-specific knowledge regarding input formats. Instead, approaches in this class generally start from a set of valid program inputs, that is, inputs that follow the format or protocol expected by the target program, and *mutate* them in order to generate new test inputs. The mutations themselves can be purely random or heuristic in nature or, in what is known as feedback-driven fuzzing, rely on information about the program execution produced by previous inputs to generate promising mutants. Furthermore, mutation operators can be generic and therefore suitable for and capable of operating on any input format, or specific to a certain input format, exploiting semantic knowledge of that format to increase mutation effectiveness. The latter category overlaps to some extent with the grammar-based input generation approaches discussed above, using knowledge of input grammars not to generate new inputs but to generate mutations that remain conformant with a given grammar.

Mutation-based fuzzing works by applying some form of mutation operator to a valid seed input in order to generate new inputs for the target system. The most straightforward form of this is simply flipping random bits in the seed input, but a wide variety of mutation operators of varying complexity have been proposed, ranging from the aforementioned random bit flips or simple arithmetic operations to input grammar aware mutation operators such as those proposed in [HHZ12]. Besides more advanced mutation operators, research on mutation-based fuzzing also deals with mutation strategies [RM11], fuzz scheduling [WCGB13, CWB15] and seed selection [RCA+14] for the maximization of fuzzing efficiency.

A mathematical model for the formalization of mutation-based fuzzing has been proposed in [CWB15], as part of an approach for choosing a mutation ratio that maximizes the probability of finding bugs. Test cases are modeled as inputs in the form of fixed-length binary sequences and execution traces are used to determine dependencies between different input bits, which amounts to a rough approximation of the syntatic structure of the input format. On this basis, the mutation ratio for a given program and seed input can be optimized to increase fuzzing efficiency.

Recent work has also dealt with mutation-based fuzzing for specific input formats, such as [JNB16], wherein the authors propose mutation operators specifically for XML inputs in order to generate inputs that are both valid and malicious as such inputs are both harder to detect than randomly mutated ones and more likely to trigger vulnerabilities in the target system.

Alongside work on guided mutational fuzzing using instrumentation and tools that attempt to utilize symbolic execution to discover new paths, this recent work on formalization and input format-specific mutation operators is indicative of a focus in fuzzing research on overcoming the efficiency limitations of blind mutational fuzzing by incorporating more information regarding the structure of the target program and its input format. As discussed above, such developments also exemplify the move towards grey box techniques in general.

Fuzzing In Practice

Fuzzing is widely applied in practice as an efficient means of vulnerability detection that requires comparatively little manual intervention. Consequently, a wide variety of fuzzing tools have been developed. They are aimed at different usage scenarios, implement different strategies and vary in their maturity, efficiency and ease of use. Guided fuzzers in particular are rarely generic but rather more commonly aimed at specific usage scenarios due to the need to observe the execution behavior of the target system. The most widely used fuzzing tools are

- *CERT Basic Fuzzing Framework (BFF)* [CERa]. BFF implements mutational fuzzing on file input for software that runs on the Linux and OSX operating systems. Using the zzuf fuzzer [Hoc] internally, BFF is intended to achieve ease of use and minimal manual intervention. As such, it includes tooling to minimize the amount of supervision a fuzzing campaign requires, can automatically perform test case minimization and crashing test case deduplication and supports the use of machine learning techniques for increasing the efficiency of fuzz scheduling and input file selection.

- *FOE*, the *CERT Failure Observation Engine* [CERb]. FOE is a mutation-based fuzzer for file input targeting applications for Windows systems, which shares many of the internals and features of BFF. CERT maintains lists of public vulnerabilities that were found with BFF[5] and FOE[6].

- *AFL* [Zal]. AFL is not a blind fuzzer but rather relies on compile time instrumentation and genetic algorithms to generate and prioritize new input variations. While the reliance on instrumentation for coverage information renders AFL a white or grey box approach rather than a strictly black box one, it also allows the fuzzer to synthesize fairly complex input formats without requiring explicit information about the input format such as a grammar specification. This allows AFL to, for instance, generate valid JPEG files essentially from

[5] `https://vuls.cert.org/confluence/display/tools/Public+Vulnerabilities+Discovered+Using+BFF`
[6] `https://vuls.cert.org/confluence/display/tools/Public+Vulnerabilities+Discovered+Using+FOE`

scratch[7]. AFL also supports test case minimization and the synthesis of test case corpora designed to maximize coverage. As the tool requires next to no manual intervention or configuration, it has been widely applied to numerous programs and has proved itself capable of discovering a significant number of vulnerabilities. AFL is generally designed for the fuzzing of user-mode programs and targets file inputs. Attempts to implement similar coverage-guided fuzzing approaches have been made in order to target lower level code, such as Linux kernel code [Gro, Goo]. Such alternative implementations or extensions highlight the applicability of coverage-guided fuzzing to a wide variety of usage scenarios.

Despite the recent increase in the popularity of fuzzing for security testing in general and AFL-style tools specifically and the accompanying advances in these tools, efficiency remains a major concern as it does in all forms of security testing. The extent and efficiency of the parallelization support offered by fuzzing tools varies, and, while correct test parallelization is not straightforward (as we discuss in Section 2.6), such issues are largely left unaddressed in practice.

2.2.2 Binary Analysis and Instrumentation

While testers in black box settings by definition do not have access to the source code of the target program, in practice binaries may be available and can be used to reconstruct information about the internal workings or structure of the target application. Such scenarios do not fall under a strict definition of black box testing but, due to the incomplete nature of the information that is generally recoverable from binaries, can not be considered as conventional white box testing either. Instead, they most closely resemble grey box testing.

Researchers have dedicated significant effort to attempts to obtain the information required to apply various white box testing techniques from the target binaries in such scenarios. The resulting techniques include attempts to apply instrumentation-guided fuzzing to binaries by utilizing Dynamic Binary Instrumentation, advanced binary analysis frameworks and techniques such as Dynamic Taint Analysis. In this section, we first discuss the use of Dynamic Binary Analysis for security testing before moving on to Dynamic Taint Analysis.

Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) is a technique that, through the injection of instrumentation code, allows the fine-grained observation and analysis of the runtime behavior of an application. Crucially, the instrumentation is transparent to the target application and performed on the binary level – source code need not be available. Typical uses of DBI include runtime profiling, optimization tools and runtime memory safety checkers.

In order to facilitate portable analyses of such instrumented binary applications on different platforms and architectures, some DBI frameworks also lift the binary code from the target application to an intermediate representation not unlike those used by compilers. The most notable example of this approach is Valgrind [NS03], a widely used tool for dynamic binary analysis.

This step of runtime lifting to a form resembling a compiler IR allows applying techniques that would usually be applied at compile time to application that are only available to the tester in binary form. Security testers can then apply techniques that require compile time instrumentation, such as AFL-style guided fuzzing to application without having access to the source code –

[7] https://lcamtuf.blogspot.de/2014/11/pulling-jpegs-out-of-thin-air.html

essentially using white box techniques in a black box setting. AFL itself includes support for an experimental binary-only mode using the QEMU [Bel05] user-mode emulation functionality.

Notably, the usefulness of the intermediate representations used in DBI tools and frameworks is not restricted to runtime instrumentation and analysis. For instance, parts of the Valgrind project have been used in binary analysis and symbolic execution tools in order to facilitate portability across architectures and platforms [SWS+16].

As DBI tools and frameworks continue to improve and the gap between working on lifted and compile-time IR continues to shrink, the availability and efficiency of security testing and analysis techniques accessible to testers in binary-only scenarios continues to improve.

Dynamic Taint Analysis

Dynamic Taint Analysis (DTA) is a technique for assessing at runtime which computations are affected by (*tainted*) previously specified sources, such as user input. Untrusted input is treated as *tainted*, and that taint is then spread to other values that are computed from tainted sources. DTA is widely used in the context of security testing and can be used to assess, for instance, whether input from an untrusted source is executed or used to compute memory addresses that are then accessed (with the latter indicating a potential memory safety issue).

DTA has been used to augment fuzzing techniques [GLR09, BBGM12], sometimes in combination with symbolic execution [WWGZ11] in order to increase fuzzing efficiency and coverage. By incorporating taint analysis, fuzzing approaches can determine which parts of the input are particularly promising targets for mutation, and, when used in conjunction with branch instrumentation, which parts of the input need to be mutated to affect certain branching decisions in order to discover new paths. This way, an increase in fuzzing coverage can be achieved without having to make use of more expensive (in terms of run time overhead) techniques, such as symbolic execution, and potential vulnerabilities can be detected during fuzzing even if they do not directly lead to an observable crash or memory safety violation.

2.2.3 Applicability

Table 2.2 summarizes the characteristics of the security testing approaches and related techniques described in this section and their applicability in the cloud context.

2.3 Model-based Security Testing

Model-based security testing utilizes, as the name implies, models to assess whether a software system possesses the security properties it is expected to. It is related to and derives from classical model-based testing, wherein explicit models of a target system and or the environment it is expected to operate in are used to algorithmically derive test cases for the target system. As with most software testing approaches, most classical model-based testing techniques focus primariliy on functional testing and are therefore not directly applicable to security testing. An overview and classification of general model-based testing techniques can be found in [UPL12].

What sets model-based security testing techniques apart from general model-based testing, apart from the focus on non-functional properties, are not primarily the test generation techniques and algorithms themselves but rather the incorporation of and focus on security properties in both

Table 2.2: Summary of black box security testing approaches and related techniques for the cloud

	Stage of Life Cycle	Artifacts required	Level of practical adoption	Available tools
Generation-Based Fuzzing	Coding or later	Binaries	Medium	SPIKE, jsfunfuzz, several others.
Mutation-Based Fuzzing	Coding or later	Binaries	High	AFL, AFLFast, zzuf, many others. Adaptations for specific usage scenarios are common (e.g. TriforceAFL, syzkaller).
Dynamic Binary Instrumentation	Coding or later	Binaries	High	Many Valgrind- or DynamoRIO-based tools, e.g. memcheck.
Dynamic Taint Analysis	Coding or later	Binaries	Low	Mainly non-public research prototypes (e.g. TaintCheck).

the model of the target system and the model of its environment. Felderer et al. classify model-based security testing techniques according to the system security model, the environment security model and the proposed test selection criteria [FZB+16], as well as providing an extensive overview of relevant research in the area of model-based security testing. In this classification, system security models can be either models of security properties (as in the aforementioned CIA triad), models of vulnerabilities (roughly comparable to fault models in dependability and robustness testing) or conventional models of functional security mechanisms. Environment security models can be either threat models or attack models.

Schieferdecker et al. [SGS12] distinguish between three different categories of models that are of relevance for model based security testing:

1. *Architectural and functional models*, which model the general behavior of the target system and are useful for functional security testing or for the analysis of interaction and protocol models that relate to security-critical interfaces.

2. *Threat, fault and risk models*, which model the causes and consequences of potential deviations from the desired behavior of the target system in the form of, as the name implies, threats, faults and risks. The attack trees [Sch99] we briefly discuss in Section 2.4 fall into this category.

3. *Weakness and vulnerability models*, which model flaws in the target system directly, for instance by applying techniques related to mutation testing to models of the target system [WAW09].

Over the remainder, we will first expand on some specific model-based security testing techniques that exhibit particular relevance to the ESCUDO-CLOUD project, before subsequently moving on to discussing the concerns regarding the practical applicability and scalability of model-based security testing approaches that have hampered the practical adoption of such techniques.

2.3.1 Model-based Security Testing Techniques

In this section, we highlight some model-based approaches to security testing that are of particular practical relevance or apply particularly well to the problems faced in the context of the ESCUDO-CLOUD use cases.

- **Model-based testing for cloud environments**. Cloud and multi-cloud settings, as used in this project, impose new challenges on security testing in general and model-based security testing in particular due to the variability and openness of the cloud environment. A technique for the model-based security testing of cloud computing environments has been proposed by Zech [Zec11]. The proposed technique first utilizes a model of the cloud system and its elements along with a repository of known vulnerabilities to perform a risk analysis, resulting in a *risk model*. This risk model forms the basis on which negative security requirements are derived, which then in turn are used to generate *misuse cases*. These misuse cases can then be used to generate executable test cases.

- **Model-based testing for cryptographic components**. Botella et al. [BBC+13] present a technique for the model-based security testing of cryptographic components. In the proposed approach, a security test engineer incrementally generates a test generation model from the component specifications and derives test purposes from the specified security testing objectives. The two formal artifacts resulting from this first step (test purposes and test generation model, modeled in a test selection language and UML/OCL, respectively) can then be used to drive an automated test generation process. The proposed approach is applicable to both software and hardware cryptographic modules and can support both functional and non-functional security testing.

- **Model based testing for security protocols**. Dadeau et al. [DHK11] present a model-based security testing technique for implementations of security protocols that utilizes High-Level Security Protocol Language (HLSPL) models. The proposed technique is essentially a form of mutation testing, whereby specifically defined mutation operators are applied to HLSPL models to introduce flaws in the modeled security protocol. Traces of attacks on the mutated HLSPL models can then be used to derive security test cases for an implementation of the protocol.

2.3.2 Practical Concerns

While model-based security testing is a promising research direction for assuring the security of modern cloud systems, there are several practical concerns that remain hinderances to its wider adoption. In particular, these are the requirements of an explicit, typically manually specified model for the system and its environment along with the corresponding cost of creating these models as well as scalability concerns. The latter in particular renders the application of such techniques to the kind of large scale cloud systems that are considered in the context of this project challenging as model-based techniques frequently do not scale to such systems. Moreover, the necessary models of the operational environment are usually missing and would require significant manual effort to create, effort that would potentially need to be (at least partially) repeated when the operational environment and its components change, which is a frequent occurence in cloud and multi-cloud settings due to the inherent flexibility of such settings.

Consequently, model-based security testing is applicable primarily to smaller subcomponents, although the requirement for an explicit model of the target system may frequently prove to be a hinderance even for such components: The effort and expertise required to manually create the required models is significant and may be better spent on security testing approaches that require less manual effort (such as fuzzing-based approaches, see Section 2.2.1) or puts the required security testing expertise to use more directly (such as penetration testing, see Section 2.4).

Finally, the effort required for and challenge associated with maintaining useful models of the flexible, frequently changing operational environment of modern, large scale, cloud-based applications underlines the importance of focusing security testing on well defined interfaces between the target component and its environment.

2.3.3 Applicability

Table 2.3 summarizes the characteristics of the model-based security testing techniques described in this section and their applicability in the cloud context.

Table 2.3: Summary of model-based security testing techniques

	Stage of Life Cycle	**Artifacts required**	**Level of practical adoption**	**Available tools**
Model-based testing for cloud environments	Specification or later	Model	Low	Process framework, no tools are publically available but academic prototypes exist for individual steps.
Model-based testing for cryptographic components	Specification or later	Model	Low	Partially manual process, individual steps may be supported by general (i.e., not specific to security testing) model based testing tools.
Model based testing for security protocols	Specification or later	Model	Low	AVISPA [ABB+05]

2.4 Penetration Testing

In the broadest possible sense, penetration testing is any form of security testing wherein the tester essentially attempts to break into a system in the same manner a potential attacker might. The concept of penetration testing is not limited to software security testing (as in the previous testing approaches)[8] and can also be applied at, for instance, the organisational level to test adherence to security procedures. Penetration tests can be classified according to the amount of information about the target system the tester has access to in advance of and during the penetration test,

[8]For example, black box testing primarily addresses automated software testing whereas penetration testing often covers manual aspects that can target more than software (e.g., organizational aspects). Hence, according to the amount of information about the target system, the penetration tests are classified as either black box or white box penetration tests.

ranging from fully informed penetration tests which essentially correspond to white box testing to uninformed penetration tests which correspond to black box testing. Due to the higher efficiency, the former approach is more common.

Over the course of this section, we first cover white box or informed penetration testing approaches, followed by black box or uninformed approaches before concluding by briefly covering penetration test automation. Note that, unlike the other approaches described and discussed here, penetration testing is not generally an automatable form of software testing (although automated security and vulnerability assessment tools, which we discuss in Section 2.5 commonly play a key role). Rather, it is an activity that commonly requires manual effort by specifically trained testers.

As noted in [McD00], penetration testing typically follows either the flaw hypothesis [Wei73] or the attack tree [SSSW98, Sch99] approach. The flaw hypothesis approach can be summarized as a sequence consisting of the following six steps:

1. Definition of penetration testing goals and scope.
2. Completion of a background study.
3. Generation of hypothetical flaws.
4. Confirmation of hypothesized flaws.
5. Generalization of confirmed flaws.
6. Elimination of confirmed flaws.

For the attack trees, as proposed in [Sch99], attacks are modeled as a simple tree with the target or goal of the attack as the tree's root. Nodes in the tree can be either conjunctive or disjunctive, with the root being the latter. Nodes in the tree represent actions. The action represented by a node can be accomplished if one (for a disjunctive node) or all (for a conjunctive node) of the actions represented by that node's children can be accomplished. Leaf nodes model actions without prerequisites.

Regular penetration testing is a key part of ensuring the security of critical systems and is required or suggested by several information security standards, such as the Payment Card Industry Data Security Standard (PCI DSS) [PCI16b] or NIST SP 800-53 [Nat11] and several guidelines on how such penetration tests should be conducted have been published [Nat08, Pen15]. Nonetheless, due to the manual effort and tester expertise required to conduct effective penetration testing, it remains the exception rather than the norm for non-critical systems and should be viewed as orthogonal and complementary to the software security testing techniques discussed throughout the rest of this document.

2.4.1 White Box Penetration Testing

As described above, white box or informed penetration tests are those in which assessors have knowledge of the internal structure and workings of their target, such as source code access in a software penetration test or knowledge of, for instance, the network layout, IP addresses, hostnames and so forth when targeting a network.

As in the software testing approaches discussed above, such a white box approach offers benefits in efficiency compared to black box penetration testing. However, it does not reveal how easy it would be for an attacker to gain information about the internal structure and workings of the

target as that information is already provided to testers and they therefore do not need to discover it themselves.

In the first of the two classical approaches to penetration testing described in Section 2.4, the flaw hypothesis methodology, the additional information available to testers can be utilized effectively in the second step, the completion of a background study, increasing the efficiency and effectiveness of the remaining steps building on the background study.

2.4.2 Black Box Penetration Testing

In black box or uninformed penetration testing, penetration testers do not have access to knowledge of the target's internal workings or structure. This more closely corresponds to the situation in which real attackers would find themselves and forces testers to attempt to obtain the information they require themselves, using techniques ranging from reverse engineering for software penetration tests to social engineering when targeting organizations. While this additional step can reveal useful information about ways in which an attacker might obtain information about the target's internal workings or structure, it also increases the penetration testing effort and consequently leads to additional costs and reduced efficiency.

The second of the two classical approaches to penetration testing discussed in Section 2.4, the creation of attack trees, maps relatively well to such a setting since attack trees can be constructed with relatively little knowledge of the penetration testing target's internal workings or structure and information acquisition can be modeled as part of the attack tree.

2.4.3 Penetration Test Automation

As discussed in Section 2.4, penetration testing requires significant manual effort and tester expertise, which in practice also entails non-negligible costs. Consequently, attempts have been made to create automated techniques that can either reduce the manual effort required for penetration tests (including tools such as vulnerability scanners, which we discuss in Section 2.5) or replace it outright with a fully automated approach offering the same or similar benefits. Due to the fact that penetration tests can include steps such as social engineering, full automation is clearly not a realistic proposition at this stage. Nonetheless, researchers have made efforts to automate parts of the process, using models of the target system and model-based planning techniques to generate simulated attacks. Hoffmann gives an overview over the background and challenges in this research area in [Hof15].

In conclusion, we note that, while tool support exists and continues to improve and automation is an active research area, thorough penetration testing is a process that requires significant manual effort and tester expertise. While penetration testing is nonetheless essential for ensuring the security of critical systems, it is consequently not always feasible for other software systems. In such scenarios, security testing techniques that are more suitable for automation and do not require the same extent of tester expertise and target system familiarity should be used as extensively as possible in order to compensate at least partially for the lack of penetration testing.

2.4.4 Applicability

Table 2.4 summarizes the characteristics of the security testing techniques described in this section and their applicability to modern cloud-based software systems.

Table 2.4: Summary of penetration testing techniques for the Cloud

	Stage of Life Cycle	Artifacts required	Level of practical adoption	Available tools
White-box penetration testing	Verification or later	Interface specifications, full source code, any other available documentation	Low	Manual process, various tools associated with other security testing techniques may be utilized by the tester.
Black-box penetration testing	Verification or later	Interface specifications	Low	Manual process, vulnerability assessment tools are commonly utilized by testers.

2.5 Vulnerability Assessment

Generally speaking, vulnerability assessment describes a process whereby vulnerabilities in a system are identified and prioritized. In the context of software security testing, as far as the vulnerability identification step of the process is concerned, the term frequently refers to the use of vulnerability scanners and other automated tools to detect potential flaws belonging to known classes of vulnerabilities. In contrast to the penetration testing approaches discussed in the previous Section 2.4, this is a much more tool-driven, automated process, although it is worth noting that many of the tools discussed in this section are also commonly used by penetration testers as part of their work. We give an overview of the different tool categories along with examples of each category next, followed by a brief discussion of vulnerability ranking and prioritization.

2.5.1 Vulnerability Assessment Tools

The first and most straightforward step when conducting vulnerability assessment of a remote system or network is typically a simple port scan to identify the services running on a given host or range of hosts. Modern tools frequently also include features that allow them to detect the operating system and network service version running on the remote host, which is a valuable step particularly if the detected operating system or application version has known vulnerabilities that are remotely exploitable. The most well known tool in this category is Nmap[9], an open source network mapping and port scanning application.

While port scanners and network mappers merely detect which hosts and services are available in a network, vulnerability scanners also automatically check if those services are vulnerable to a set of known exploits. Vulnerability scanners may also check for common misconfigurations, such as SMTP servers that are accidentally misconfigured as open mail relays. While little manual effort and testing expertise is required on the part of the users of vulnerability scanners, the automatic tests for known exploits and misconfigurations require significant effort to keep updated and maintained. Thus, commercial tools in this category, such as the Nessus Vulnerability Scanner[10], may offer access to up-to-date versions of these checks on a subscription basis.

There are also more specifically targeted tools for certain common use cases available, such

[9] `https://nmap.org/`

[10] `https://www.tenable.com/products/nessus-vulnerability-scanner`

as the automated security testing of web applications at various stages of their development. Such tools contain additional functionality specifically aimed at web applications, such as crawlers to explore the web application, proxies to observe manual tester interaction with the target web application and specific checks for common web application vulnerabilities like cross-site scripting or SQL injection. The degree of automation offered by such tools, the amount of tester expertise required and the stages of the target web application's development at which they are suitable for use varies. A well-known example of this category is the open source tool OWASP Zed Attack Proxy (ZAP)[11] (detailed information is described in Section 2.7 and Chapter 3).

Finally, there are also fully featured, integrated penetration testing software suites available. These typically integrate various other vulnerability assessment tools and are designed to aid penetration testing by reducing manual effort where applicable. Note that tools in this category do not fill the same role as the research area of penetration testing automation that we briefly discussed in Section 2.4 and that these tools still require significant manual effort and penetration testing expertise on the part of the testers. The Metasploit Framework[12], an exploit development framework, falls under this category as does Kali Linux[13], a Linux distribution based on Debian and aimed specifically at penetration testers. Kali includes numerous security and penetration testing tools, including the Metasploit Framework and Nmap.

2.5.2 Vulnerability Rating and Prioritization

To deal with vulnerabilities in software components, whether it is by expending development effort to fix them or by deploying mitigation techniques for vulnerabilities in third party components, administrators and developers alike must first understand the *impact* of a vulnerability, that is, the ease with which a potential attacker might exploit it and the extent of the information such an attacker might obtain and the damage they might be able to inflict on the vulnerable system. In this section, we briefly discuss what vulnerability ranking and prioritization entails, what approaches exist and what challenges are associated with this process.

The best-known vulnerability severity assessment approach is the *Common Vulnerability Scoring System* (CVSS) [FIR15], which is used by, among others, the National Vulnerability Database (NVD)[14] and the CERT Coordination Center[15] to rank the severity of software vulnerabilities. In CVSS v3.0, the current version of the standard, vulnerabilities are ranked according to metrics belonging to three different groups:

- *Base* metrics capture those attributes of a vulnerability that are neither affected by the operational environment nor changeable over time. This group is further split into impact and exploitability metrics, with the former representing the consequences of a successful exploitation of the vulnerability, that is, the extent of the damage an attacker might do or the information they might obtain, and the latter representing the ease with which a potential attacker might be able to exploit the vulnerability.

- *Temporal* metrics represent attributes that may change over time but are unaffected by the operational environment of the vulnerable component, such as the *Remediation Level*, which

[11] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[12] https://www.metasploit.com/
[13] https://www.kali.org/
[14] https://nvd.nist.gov/
[15] https://www.cert.org/

would be affected by the availability of an official patch for the vulnerability but is independent of the operational environment.

- *Environmental* metrics represent attributes of the vulnerability that are dependent on a particular operational environment, such as modifications of base metrics, for instance due to the presence of mitigation mechanisms that affect exploitability.

Analysts score vulnerabilities according to metrics belonging to each group, resulting in a numerical score in the range 0 to 10, with 10 representing the highest vulnerability severity.

It has been noted [FM09] that relying on an assessment based only on the most widely used of the three metric groups, the base metrics, is problematic as it fails to take the context in which a vulnerable component is operating into account. Incorporating temporal and especially environmental metrics can improve the accuracy with which the vulnerability score represents the actual severity of a vulnerability for a particular organization, which in turn helps in prioritizing vulnerabilities for fixing or mitigation. However, making use of the additional metric groups, particularly the environmental metrics, also imposes additional effort and cost as analysts need to customize their scoring for different operational environments and potentially rescore vulnerabilities over time as temporal metrics change.

In practice, then, it is not uncommon for users and administrators to exclusively consider the base metrics, in which case the resulting severity estimate may be inaccurate or even misleading. Furthermore, vulnerability impact ratings may initially underestimate the severity of a vulnerability [AAD+09] and rankings do not cover potential interactions between multiple flaws or vulnerabilities resulting in increases in exploitability or impact.

2.5.3 Applicability

Table 2.5 summarizes the characteristics of the security testing techniques described in this section and their applicability in the cloud context.

Table 2.5: Summary of vulnerability assessment techniques for the Cloud

	Stage of Life Cycle	**Artifacts required**	**Level of practical adoption**	**Available tools**
Network mappers and scanners	Verification or later	Operational target system (possibly in a segregated test environment)	High	nmap, numerous other tools, both open source and commercial, are available
Vulnerability scanners	Verification or later	Operational target system (possibly in a segregated test environment)	Medium	Various commercial tools, e.g. Nessus Vulnerability Scanner
Web Application scanners	Verification or later	Operational target system (possibly in a segregated test environment)	Medium	OWASP ZAP and others

2.6 Test Parallelization for Security Testing

Throughout our discussions of various approaches and techniques for security testing in this chapter, we have pointed out efficiency concerns where applicable. We have also noted the need, particularly in modern cloud and multi-cloud systems, to focus testing efforts on interfaces. As discussed in Section 2.2.1, security testing is quite closely related to robustness testing[16] (with fuzzing, one of the most popular software security testing techniques today starting out as a dependability testing approach).

For a robustness testing, a perturbation analysis is needed to investigate how a target system, or parts of the system, behave under anomalous (i.e., perturbed) operational conditions. In practice, perturbation analysis simulates various scenarios to represent deviations from standard system specification (also called "misuse cases"). The underlying assumption is that those anomalous cases have not been taken into account during the system designing stage and the corresponding reactions might not have been specified. Contrary to traditional functional testing (correctness testing)[17] and penetration testing[18], the primary target of perturbation analysis is to assess system robustness. It is important to mention that perturbation analysis is not an approach to determine system correctness, but primarily to assess the performance of robustness/fault-tolerant mechanisms of the target system when encountering perturbations.

Thus, while robustness testing techniques like Software Fault Injection (SFI) are used to experimentally assess the robustness of software systems against faults arising from hardware devices, third-party software components, untrusted users and other sources, and software security testing techniques like fuzzing are intended to assess the resilience of such a system against deliberately malicious actors, due to the randomized nature of fuzzing in particular, the overlap is immense. This also means that many of the same concerns about testing efficiency and throughput apply in both scenarios.

Given the high complexity of modern software, robustness and security testing alike typically require a significant number of experiments to cover all relevant cases for the validation of fault-tolerance or security mechanisms, with studies reporting thousands, or even millions, of injected faults [KD00, AFR02, DLAS+12, NCDM13] or mutated inputs [Oeh05]. The problem of high experiment counts is exacerbated by evidence that *simultaneous perturbations*, i.e., combinations of several injected faults or several corruptions in the mutated input, need to be considered as well. Recent studies [GDJ+11, JGS11] show that failure recovery protocols in distributed systems exhibit vulnerabilities to simultaneously occurring faults and can, hence, only be uncovered by injecting fault combinations. A "combinatorial explosion" of the number of experiments is the consequence. Similar findings were obtained in recent work on operating systems and software libraries, which showed that software faults cause the simultaneous corruption of several interface parameters as well as shared memory contents [LNW+14] and that simultaneous corruptions can uncover robustness issues which would not be found by singular corruptions [WTSS13]. Fuzzing tools have similarly proven more capable of uncovering security issues when generating higher order mutations of program input, and subjecting previously mutated input to further mutations is commonplace in widely used fuzzing applications, including the state of the art fuzzer AFL [Zal].

Despite the demonstrated utility of simultaneous fault injections and higher order input muta-

[16] Refers to the correctness of implementation (in particular referring to availability and integrity) in the presence of failures.

[17] Usually a stable architecture and source code level implementation details are needed

[18] Requires highly skilled people and can cause interruption of network services

tions, the combinatorial explosion of the number of experiments remains a considerable challenge for their applicability.

In order to cope with the high number of experiments or test cases, two different strategies can be adopted. The first strategy attempts to reduce the number of experiments that need to be performed. Search-based techniques and sampling strategies for large test sets (e.g., [JH08, SAM08, JGS11, NCDM13]) fall into this category. The second strategy attempts to utilize the increasing computational power of modern hardware, where several experiments are executed at the same time on the same machine (*parallelization*) to better utilize the parallelism of the underlying hardware (e.g., [Las05, DCBM06, OU10]).

While parallelization ("throwing hardware at the problem") is less elegant, it is an appealing solution since it is generally applicable, whereas the applicability of sampling and search-based techniques depends on domain-specific knowledge in most cases. Parallelization, therefore, seems to be a promising solution to cope with the high number of experiments and test cases, especially as it can be combined with domain-specific sampling and pruning.

Nevertheless, parallelization relies on an implicit assumption that *executing several experiments in parallel does not affect the validity of results*. We hypothesize that this assumption is not trivial. Even if the experimenter takes great care to avoid interference between experiments (e.g., by running them on separate CPUs or virtual machines), there is a number of subtle factors (such as resource contention and timing of events) that can change the behavior of the target system (e.g., faults can lead to different failure modes than those observed under sequential execution), thus invalidating the results and negating the benefits of parallelization. This is a concern especially for embedded, real-time, and systems software, which are an important target of fault injection experiments and security testing alike, and where studies have shown that faults often exhibit non-determinism and time-sensitive behavior [CGN+13, AFR02].

A straightforward way to avoid such concerns would be to simply scale up the number of machines used for test execution, parallelizing tests not by executing multiple test cases on the same machine at the same time but by executing N different test cases on N different machines. This straightforward approach to test parallelization, illustrated in Figure 2.3 with a centralized control instance, sidesteps concerns about test interference or threats to result validity by avoiding parallel test execution on each individual instance, but as a result it is highly inefficient, in terms of resource utilization as well as cost. A solution that allows more efficient hardware utilization and consequently lower costs and better testing efficiency is highly desirable.

Hence, in order to conduct *efficient and accurate* parallelization, we have developed PArallel fault INjections (PAIN) as a framework for executing perturbation tests, such as software fault injection (SFI) experiments, in parallel. As SFI, much like extensive security testing, is applied mostly for the assessment of critical systems, a major concern that outweighs performance considerations is the confidence in the validity of the experimental results; it is of utmost importance to avoid interference of PAIN experiments that affects their outcome. In addition to experiment throughput, we therefore also need to assess the validity of results from parallel experiments. To this end, we have developed an experimental environment for the study of parallel perturbation tests and similar system-level tests, including fuzzing-style security testing. Furthermore, we provide guidelines to tune the main factors that affect experiment throughput and the validity of PAIN experiments, including the degree of parallelism and failure detection timeouts. These guidelines are based on a qualitative and quantitative analysis of the impact of parallelism on the result validity and experiment throughput of extensive fault injection experiments that we have conducted on the Linux kernel under an Android emulator environment. For a more detailed discussion of the

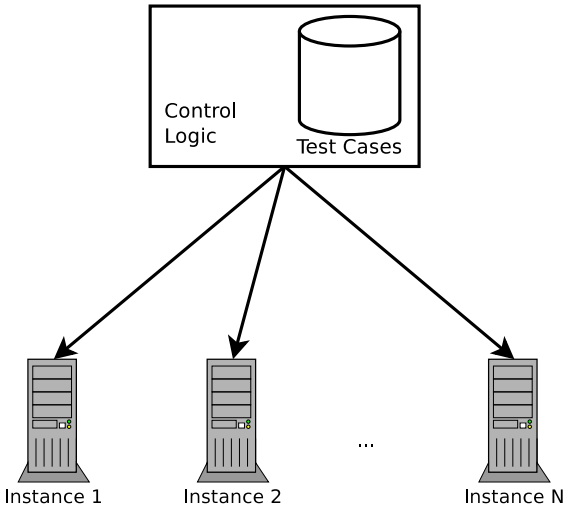

Figure 2.3: Simple test parallelization using separate machines

experiment evaluation and the corresponding results, refer to [WSN+15].

Essentially, there are two key questions that need to be addressed:

1. Can parallel execution of multiple software fault injection experiments or security testing test cases on the same machine increase the throughput of the robustness or security testing?

2. Do the results obtained from such executions differ from those obtained with sequential executions of the same experiments or test cases?

If the answer to the first question is positive and the answer to the second question negative, then perturbation test parallelization has no adverse effects and should be applied whenever parallel hardware is available. However, if the answer to the first question is negative and the answer to the second question positive, parallelization should be avoided. If both answers are negative, the decision whether to parallelize or not should be driven by other factors, such as hardware cost or complexity of the experiment setup. If both are positive, parallelism can be beneficial, but it can also potentially affect the accuracy of results. In this case, we need to investigate a third question:

3. Can the parallelization of experiments or test cases be tuned to achieve the desired increase in throughput while avoiding inaccuracies in the obtained results?

As discussed in [WSN+15], experiments involving the parallelization of perturbation tests with corrupted Linux kernel modules on emulated Android systems showed the answer to both of the first two of the questions laid out above to be positive, that is, parallel execution of perturbation tests increases throughput and affects the results compared to sequential execution. Moreover, the observed changes are the most pronounced for those experiments that result in two failure modes that depend on timeouts for detection. This result is both highly relevant and directly applicable to security testing as well, since numerous security testing approaches and tools, most notably those falling into the fuzzing category, including AFL [Zal], the current state of the art fuzzer, rely on timeouts to detect hangs of the target program. Consequently, we suspect that the increased rates for parallel experiments are false positives of these detectors and that their timeouts need adjustment in the parallel case. This, then, introduces a tradeoff, as laid out in the

third question posed above: High timeout values negatively affect throughput whereas low timeout values threaten result accuracy by way of false positives.

A naïve approach to avoid these false positives would be to increase the timeout values with the number of parallel instances. However, as the degree by which execution times increase with the numbers of instances is generally unknown and also depends on the execution platform (hardware and OS), this entails an iterative process of trial and error until suitable timeout values are found.

A better strategy is to estimate values based on observations made during so-called *golden runs* on the intended execution platform *without perturbations*. Such runs should be performed for each targeted level of parallelism and relevant timing data be recorded as a baseline to derive suitable timeout values. However, while such an approach can result in a clear improvement over constant, unadjusted timeout values, differences between the timing behavior exhibited during calibration runs and actual experiment runs can nonetheless lead to significant levels of false positives.

Since, as the results in [WSN+15] show, a dedicated calibration setup may exhibit different timing behavior than real test executions, we conclude that timing data from real perturbation tests should be used for choosing suitable timeout values when applying parallelization.

Using the 99.99 percentile of a distribution fitted to the timing data from actual perturbation tests, we obtained a number of hang detections comparable to the original perturbation tests with tripled timeout values, confirming the suitability of our approach. With timeout values of comparable or even better accuracy than the previous trial and error approach, our systematic approach to calculating timeout values is preferable if it results in acceptable overhead. We illustrate the resulting process for test parallelization using a single machine in Figure 2.4. Since, despite the efficiency gains that intelligent test parallelization entails, utilizing multiple machines to execute test cases is nonetheless necessary to enable the testing of large scale software systems within a reasonable timeframe, we also illustrate the setup that results from applying the PAIN process in such a scenario in Figure 2.5. In such a scenario, a straightfoward realization of efficient test parallelization can be achieved by applying the adjusted timeout values obtained with PAIN to each machine and splitting the set of test cases between the available machines.

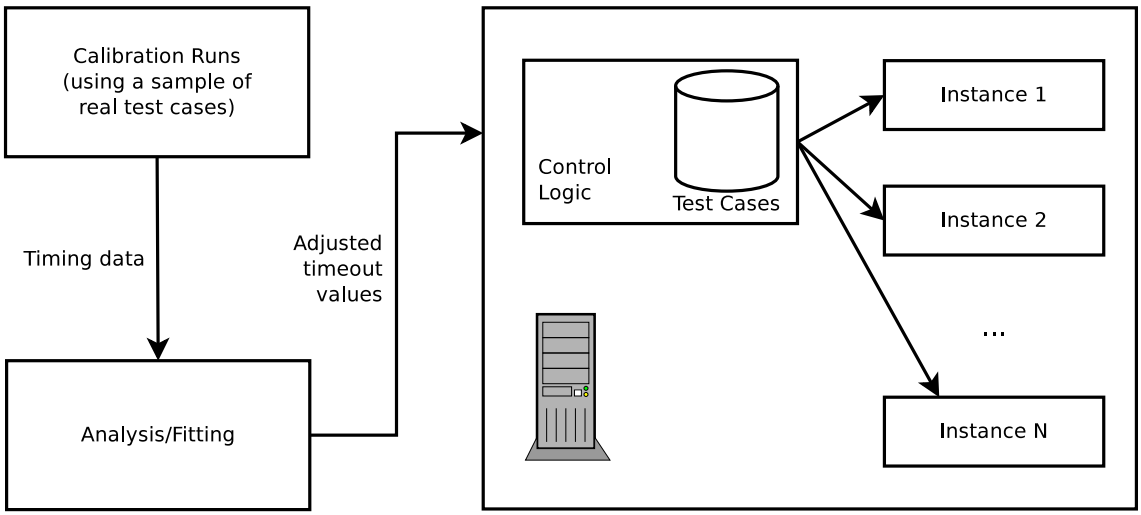

Figure 2.4: PAIN test parallelization on a single machine

As software systems become more complex and, at the same time, tend to exhibit sensitivity to complex perturbations or fault conditions [GDJ+11, JGS11, WTSS13, LNW+14], the number

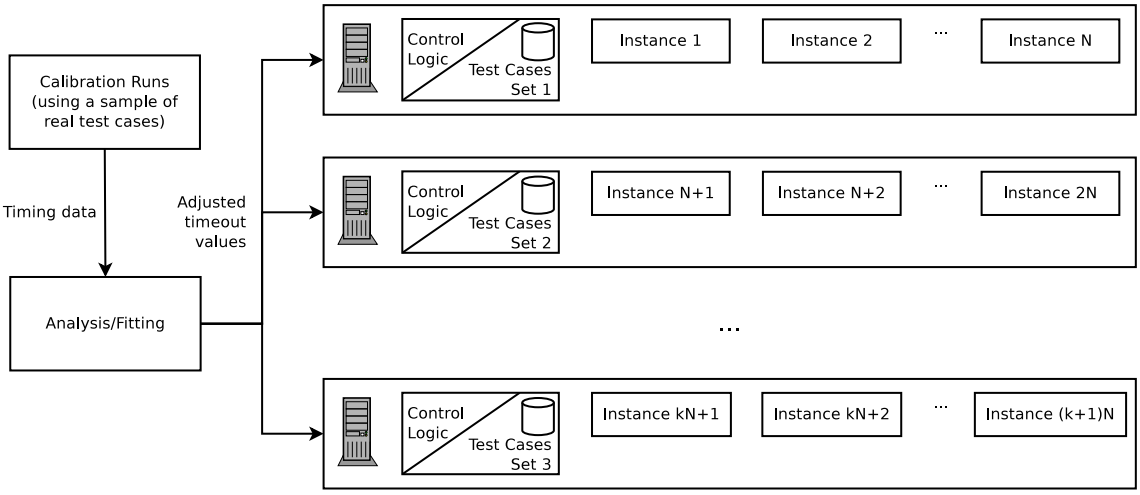

Figure 2.5: PAIN test parallelization using multiple machines

of relevant fault conditions and input mutations to test against also increases drastically. The simultaneous execution of such tests on parallel hardware has been advocated as a viable strategy to cope with rapidly increasing test counts. We have addressed the question whether such strategies can be applied to speed up perturbation tests by performing PArallel INjections (PAIN). Besides assessing the speedup of experiment throughput, we address the question whether PAIN affects the accuracy of such perturbation tests.

Our results show that while PAIN significantly improves the throughput, it also impairs the accuracy of the test results. We found that result inaccuracy is related both to the degree of parallelism and to the choice of timeout values for failure detection, due to resource contention and timing of events that influence the tests. Therefore, we provide guidelines to tune the experiments using data from preliminary test executions, in order to achieve the best test throughput while preserving result accuracy.

2.6.1 Applicability

Table 2.6 summarizes the characteristics of the test parallelization techniques described in this section and their advantages and drawbacks.

Table 2.6: Summary of test paralellization techniques

	Advantages	**Drawbacks**	**Level of practical adoption**	**Available tools**
Naïve parallelization	Simple to implement, no additional tools or analysis required	Validity of obtained results is questionable	Medium	None required
PAIN-based parallelization	Preserves result validity while improving throughput	Requires additional analysis steps	Low	PAIN

2.7 Guidelines, Methodologies and Tools

Since software security remains challenging and the consequences of breaches can represent significant business risks, but many organizations are lacking the internal expertise to develop extensive, appropriate security testing guidelines, numerous groups have proposed general methodologies and guidelines that organizations can adopt (with adjustments to their specific situation) in order to improve the security of the software they build. In this section, we will briefly discuss two such guidelines: The OWASP (Open Web Application Security Project) Testing Guide [MM14] and the Open Source Security Testing Methodology Manual (OSSTMM) [HB10]. Finally, two tools have been developed within ESCUDO-CLOUD for carrying out perturbation analysis based on fault injection processes.

2.7.1 OWASP Testing Guide

The Open Web Application Security Project (OWASP) is a nonprofit organization that aims to improve the state of web application security by creating educational resources and guidelines for developers and organizations. The organization is perhaps best known for the OWASP Top Ten[19], a list of the ten web application security risks the organization deems the most critical, complete with example vulnerabilities and attacks as well as guidance on how to avoid the issue. OWASP also provides several extensive guidelines on web application security, including the OWASP Developer Guide, Code Review Guide and the Testing Guide which we focus on here.

The OWASP Testing Guide is an extensive guide to web application security, starting with an overview of the scope, pre-requisites and principles of web application security testing and the role of security testing throughout the software development life cycle. The bulk of the guide is made up by an extensive overview of web application security testing objectives, techniques and tools (including guidance on the effective usage of some of the security testing tools discussed in this report, such as those in Section 2.5). In particular, it contains guidance on how to test for classes of vulnerabilities that are especially common in or specific to web applications, such as flaws in input validation or session management on the server side or cross site scripting on the client side.

2.7.2 Open Source Security Testing Methodology Manual

The Open Source Security Testing Methodology Manual (OSSTMM) by ISECOM[20] is a manual that aims to provide a complete methodology for assessing security. Unlike the OWASP Testing Guide discussed above, the OSSTMM is not a software testing guideline. Rather, it aims to deal with overall operational security and thus also addresses other operational channels, including human factors and physical security. Software or more generally computer system and network security issues are also covered, but they are neither the sole nor the primary focus of the guideline. Consequently, much of its content lies outside of the scope of this document. It underlines, however, that ensuring security is not limited to software security testing, and that a full security audit needs to take all channels that might be available to attackers into account.

Particularly in modern cloud and multi-cloud settings, it is not always easily possible for customers to determine the extent of the attack surface or to evaluate the quality of the security measures taken by the cloud service provider with respect to such channels as physical access. In such settings, full security audits can be valuable in addressing customer concerns over data security.

[19] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[20] https://www.isecom.org/

We draw two key conclusions at this point: First, software security alone is insufficient for ensuring system security, and consequently, software security testing alone is insufficient for assessing overall system security. Consequently, any approach that neglects other, conventional penetration testing and security auditing steps in the pursuit of automation may face vulnerabilities by way of other channels. Secondly, in modern cloud and multi-cloud settings, customers may not have enough control over how the infrastructure they rely on is operated to ensure that it is not vulnerable to attacks by other channels (such as physical or organizational). Consequently, the components that they have full control over, most importantly the software they are deploying, need to be hardened to be as resilient to such attacks as possible. In practice, this generally means treating infrastructure and third party components as untrusted. This, in turn, also has implications for software security testing: With little to no control over the infrastructure and frequent interactions with interchangeable third party components, access to which for testing might be limited, security testing necessarily needs to focus on the interfaces to those components.

2.7.3 Developed Supporting Tools

The following two developed tools (GRINDER and PAIN) support the fuzzing based test injection processes. While both the tools have been developed primarily with a dependability testing perspective, both the tools support the basic fuzzing processes for both dependability and security testing.

- As each use case differs in its implementation, GRINDER helps develop a common base for portability of the test injection processes to different environments.
- PAIN supports the test acceleration aspects discussed in Section 2.6

Both of the developed tools are available for public use:

GRINDER (GeneRic fault INjection tool for DEpendability and Robustness assessments) GRINDER is a fault injection testing framework that has been developed with the goal of easy adaptability to various injection targets and scenarios. While a large number of different fault injection tools exist that all target injections of specific fault types into specific locations of specific target systems (i.e., they only work for a very specific experimental assessment), GRINDER can be reused across a large variety of different systems. To support different target systems, GRINDER introduces an abstract interface to target systems. For each target of a fault injection campaign, a controller extension for this target has to be implemented and linked with GRINDER, so that GRINDER can initialize, reset, and stop the target and trigger experiment executions. To demonstrate its versatility (for both dependability and security testing) we have successfully used GRINDER in experiments with the Linux-based Android kernel and the safety mechanisms in a commercial AUTOSAR distribution. Our experience with GRINDER's usage and code reusability for these scenarios is documented in the ESCUDO-CLOUD publication [WPS+15]. GRINDER's source code is available on github under the AGPL v3 license [21].

PAIN: A framework for PArallel fault INjection experiments As already mentioned in Section 2.6, injection experiments (e.g., fuzzing injections) are a common approach to assess the robustness of component-based software systems or, more precisely, the absence thereof. In order to

[21] https://github.com/DEEDS-TUD/GRINDER

find out if adverse behavior of less critical components affects the reliable operation of more critical components in the software system, tests/faults are injected into the former and the behavior of the latter is observed in their presence. As these tests are usually conducted on fully integrated systems, individual fault injection experiments tend to have long run times. PAIN (PArallel fault INjection) exploits parallel hardware to improve on fault injection experiment throughput by executing multiple experiments concurrently. PAIN is based on the GRINDER injection framework. To demonstrate throughput improvements, we have conducted injections on the Linux-based Android OS kernel. Although experiment throughput was greatly improved by concurrent executions, we also observed significant deviations in the obtained result distributions, i.e., concurrent experiment executions led to different results than sequential executions. We identified time-sensitive failure detectors as a major cause for the observed deviations and proposed a systematic timeout calibration strategy for their elimination. The details of our study have been published in the ESCUDO-CLOUD publication [WSN+15]. All source code and configurations used in our study have been released on github under the AGPL v3 license [22].

2.8 Summary of security testing techniques

This section evaluates the security testing techniques groups described in previous sections. The analysis is done according to two parameters:

- **The resources required**. Some techniques require more resources than others to be able to apply them. For example, some white box testing approaches, such as symbolic execution-based approaches, can be very computationally extensive and consequently require significant resources both in terms of time and required hardware to execute, whereas other techniques, like many static analyses, can be comparatively cheap to execute.
- **The stage of the life cycle where it is applied**. Not all the security testing techniques can be applied in all stages of the cloud service provision. Some of them can only be used at implementation time (for example, those that require the availability of the code). Others are only suitable to be applied once an operational system exists, such as vulnerabiltiy assessment techniques.

Table 2.7: Security testing techniques over the cloud provisioning life cycle

Resource demand	Specification	Design	Coding	Verification	Operation and maintenance
High	Traceability (out of scope in ESCUDO-CLOUD)	Model-based	White/Grey box (e.g. symbolic execution, instrumentation-guided testing)	Black box (e.g. fuzzing)	Penetration testing
Low	Traceability (out of scope in ESCUDO-CLOUD)	-	White box (e.g. static analysis)	Black Box (e.g. DBI-based checks)	Vulnerability assessment

Table 2.7 describes the mapping between the techniques described and the two aforementioned parameters.

[22] https://github.com/DEEDS-TUD/PAIN

We stress the importance of utilizing intelligent test parallelization approaches in order to increase security testing efficiency while maintaining result validity. Utilizing such parallelization approaches can render computationally expensive security testing techniques feasible in scenarios where they otherwise would not have been.

The availability of the required resources and the stage of the cloud service life cycle determine the applicability of the various security testing techniques. We use this information in Section 4 to evaluate the requirements of the system. The evaluation (i.e., the potential impact, the resources involved) allows us to identify the stage of the life cycle where it applies and determine the availability of the resources. With that information we can clearly identify the techniques to use for every requirement gathered from the four use cases of ESCUDO-CLOUD (resulting from the activities carried out in WP1).

3. Application Level Verification Techniques

Cloud computing offers great opportunities for data owners to access scalable, cheap storage and compute resources to manage and protect their data. However, without adequate trust and strong integrity assurance, the challenge remains to provide the security levels expected by data owners to confidently move their assets to a CSP. Integrity checking of applications becomes critically important as data owners move their data to the cloud, entrusting the CSP, and applications running on the CSP (whatever the model - SaaS, PaaS, IaaS), with the security of that data. While it is important to first ensure the confidentiality and integrity of the data stored at the CSP, the applications responsible for the management of that data must also be verified to ensure they are neither compromised nor corrupted. In addition, the integrity of the virtual machine on which the application is running must be monitored to ensure the integrity of the application environment.

CSPs have introduced security mechanisms and policies over recent years to to secure their systems, in order to minimize the threat of attacks (both external and insider), and reinforce the confidence of customers. For example, they protect and restrict physical access to their datacentres, implement accountability and auditing processes, and apply strict access controls over critical components in the infrastructures (e.g., separation of duty principles) [PPB03]. The threat remains however that insiders with administrative power over software or hardware in the CSP can override these security mechanisms to access customer VMs. Therefore, to provide further assurances for the confidentiality and integrity of customer VMs, applications and data, new solutions need to be implemented that allow the customer to verify the status independently.

A typical scenario for an insider attack on a VM, or application running on a VM requires that the attacker have administrative capabilities over the physical infrastructure of the CSP. This can potentially, depending on the security policies enforced, enable a single rogue administrator to access the memory of executing VMs. Assuming the attacker can also gain root privileges on the VM, it is possible then to perform attacks running malicious software or manipulating user applications.

The following sections will highlight some of the techniques available for ensuring the integrity of applications through their development, distribution and use, including verification of cloud/web applications and their interfaces. Section 3.1 will first cover the processes recommended for the application development environment and how to integrate mechanisms for the verification of the application by the application consumer. Section 3.2 then covers the techniques and recommendations for ensuring the security of applications and for monitoring their integrity to prevent malicious modification. Without a secure virtual environment, subsequent measures to mitigate risks to the integrity of applications could be compromised or remain undetected for an extended period of time. Section 3.3 introduces the concept of *Trusted Computing*, covering a range of technologies for establishing a root of trust in hardware that provides a chain of trust up to the OS and to the application itself.

3.1 Code Integrity

The first step in considering the integrity of an application is to ensure that processes are in place for protecting code integrity. As a pre-requisite, this includes steps for code classification and inventory management. Code classification is an assessment that allows application developers to tier code into different levels to which security policies can be applied. Source code classification requires analysis of the content of source code which is organized into distinct components or modules to determine its inherent level of sensitivity. This process takes into consideration the relative values of three information attributes as they pertain to the content of the source code undergoing analysis. The following list defines these attributes:

- Confidentiality: The degree to which source code must be protected from unauthorized disclosure that could negatively impact the developer.
- Integrity: The degree to which source code and binary code must be protected from unauthorized modification that could negatively impact customers.
- Compliance: The degree to which source code requires legal review prior to being shared or disseminated externally. Compliance primarily applies to source code that contains regulated material (e.g., export restricted content such as cryptographic source code) or requires government notification or approval to disclose or distribute.

The Classification Tiers are defined as follows:

- Tier 1: A segregated class of source code that is highly sensitive in nature and requires additional protection measures to ensure the integrity of the products it relates to. Tier 1 source code is a subset of source code with high integrity and high confidentiality information attributes which is identified after undergoing a comprehensive review with a security team.
- Tier 2: Source code that is less sensitive in nature than Tier 1 source code and is not generally provided to customers or broadly shared with partners.
- Tier 3: Source code that is generally provided to customers or broadly shared with partners as part of standard business practices.

For example, *Tier 1* code is considered highly sensitive and confidential, requiring more stringent security policies to restrict access. An architectural consideration for Tier 1 code is how to deal with integration into part of a product. Are there, for example, some significant restrictions on what activities can be performed in certain geographies based on the tiering plus the risk associated with the build and development locations for the code?

One of the steps/activities which we would pursue with Tier 1 code is to look at *code modularity*:

- Can the highly sensitive material be segregated from the rest of the code base?
- What is the classification of the base without the Tier 1 code in it?

Inventory management allows developers to track the resources within their environment. It is not possible to maintain, let alone secure, that which you do not know exists. This includes not

only high level server information but a fine grained inventory of physical, virtual, application and OS layers as well as the relationships between them. The process requires continuous refreshing via either manual or systematic/automated processes to keep the inventory accurate. From a personnel perspective, clear responsibilities need to be established for who owns what layers and how they interact with each other.

Virtualization adds a layer of complexity for high value assets, like source control systems or repositories, which many people do not consider. There needs to be a standard relationship mapping for which physical devices host what virtual environments, and what OS and applications run on those physical and virtual machines. The critical issue arises when you start to consider the portability of VMs. Tools like vMotion are adept at keeping the environment performance optimal[1]. However, the potential exists where you could vMotion a high value system out of a secure environment if you do not add the vMotion/vCenter relationships into your architectural planning. Essentially, the software takes a VM off of a compliant physical unit for performance (or some HA task) and moves it to a physical host which may have the bandwidth to ensure performance but it is a non-compliant environment, lacking basic and necessary security controls to ensure data security.

Authentication and access control mechanisms require attention too, as these are the tools used to control *who* ultimately has access to product code. Authentication services are available today for both Windows and Linux systems tying into Active Directory (AD) systems. It is also possible to connect applications to a Single Sign On (SSO) infrastructure for improved user experience. AD groups for access control are created and the user management, including re-attestation of 'proper access', can be done as scheduled tasks. Periodic review of user access to systems and their access level should be a requirement across organisations, making it easier to manage both personnel and privilege. To that end, AD Groups can provide better granularity for control as you can segment users by role, team, function or geography. Employee terminations and the removal of access to the systems can be configured to automatically occur when an account is disabled in the corporate AD. Additionally, from an auditing standpoint, these steps demonstrate that an organisation has well defined and controlled processes for its sensitive material.

3.1.1 Build Process Integrity

A Release Manager (person responsible for ensuring the integrity of the product code) is responsible for ensuring that the build scripts that produce the final product components for customers are in a system that is protected. They are also responsible for ensuring that write access to the build scripts is limited only to those individuals involved in building the components. The Release Manager also has the power to authorize access to scripts or build jobs which are used to produce the final release build. These scripts and build jobs must be protected with write access limited to individuals involved in building components as determined by the Release Manager. By restricting read access to the scripts used to produce the final product, the developer limits the ability of an attacker to infer weaknesses inherent in a product based on build process such as flags used by a compiler. By restricting write access to the scripts used to produce the final product, attackers are limited in their ability to modify the build process and inject code from a non-authorized location, thereby bypassing review and approval processes designed to protect against the threat of malicious code. This infers a requirement to also restrict access to official build hosts to Users

[1] `http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-vsphere-evc-performance-white-paper.pdf`

approved by the Release Manager.

3.1.2 Code Signing

In order to help guarantee the integrity and authenticity of code published by a developer, product teams should use code signing to create a digital signature that allows customers to trust the products they are installing and using, and ensure they are valid products. By performing code signing, the ability of a malicious entity substituting product code with their own is limited. Code signing adds a digital signature to product artefacts (e.g., drivers, binary files, or configuration files) to authenticate the origin of the artefact and provide a claim of integrity by the author. Signing code involves more than simply invoking a command as part of the build process.

Digital certificates and associated public/private key pairs used to sign product binary code in order to confirm authenticity and integrity. A code signing certificate comprises a set of data that completely identifies an entity, and is issued by a certification authority only after that authority has verified the entity's identity. The data set includes the entity's public cryptographic key. When product binary code is signed with a private key, the recipient can use the corresponding public key retrieved from the certificate to verify the identity of its publisher.

Binaries can be signed with embedded digital signatures. Other files (as well as binaries) can be signed in the Windows environment by using a signed catalogue (a .cat file) that contains hash values and is by itself signed with an embedded digital signature. Signed catalogues are an example of detached signatures, where the content signed and the signature itself are not in the same file (the catalogue contains a hash of the content to be signed and is by itself signed).

3.2 Application Integrity

In addition to the steps outlined in the previous sections for ensuring code integrity during development and distribution, the section will outline some general recommendations for ensuring application integrity.

Application hardening techniques should be implemented to protect the application, such as encryption, obfuscation, or binary modification. These approaches can secure applications from reverse engineering and application tampering, and increase the difficulty in gaining unauthorized access. Encryption and obfuscation are considered to be baseline techniques to be used for all applications. For higher sensitivity applications however, further techniques should be implemented such as runtime tamper detection (e.g., checksums), runtime environment checks (root and debugger detection), self-healing during runtime (e.g., replace tampered code with original code), etc.

When evaluating a plan for application hardening, it is important to consider all aspects and associated components of the application (e.g., What OS is it running on? What 3rd party applications or libraries does it depend on?). Changes in these components can negatively impact the security of the application to be protected. Updates to an OS, browser or plugin could open vulnerabilities in the application environment that can be exploited. Even security patches can potentially expose the application to a new vulnerability, particularly when patches are rolled out quickly without an adequate patch configuration and testing plan in place.

Runtime detection of anomalies in the execution of an application or its environment have already been identified as an approach for ensuring the integrity of the application. The analysis these components can be broken down into three categories:

- Static Analysis - The testing and evaluation of an application by examining the code or binary without executing the application. This approach detects security vulnerabilities by analysing the application and/or its source code directly. The benefits of static analysis provide the best return when integrated into the development process using automated tools to achieve a high level of code coverage.
- Dynamic Analysis - The testing and evaluation of an application during runtime. This approach typically automates penetration testing processes to test the application against a variety of attack scenarios enabling it to uncover vulnerabilities not easily detected via static analysis techniques.
- Hybrid Analysis - Combines two approaches to improve on dynamic and static analysis providing improved accuracy and detection rates; 1) dynamic analysis is executed against the target application; 2) analysis is run on the target server.

Finally, software composition analysis techniques can be used to review the composition of software source code and binary files, identifying embedded libraries or sections of code that have known vulnerabilities. As it will be highlighted in Section 3.2.1, there are a number of recommendations from standards bodies such as PCI and OWASP that include explicit requirements for addressing component level vulnerabilities. A number of security vendors have commercial offerings that enable developers to detect and remediate any vulnerabilities that are known to exist in third party libraries and open source code included in their application.

3.2.1 Web Application Security

Web application vulnerability scanning performs a process called *spidering* to identify the pages within the application. Each page, and the fields within the application, are tested to uncover common weaknesses such as:

- Cross-site scripting (XSS)
- HTTP response splitting
- Cross-site request forgery (CSRF)
- Clickjacking
- Disclosure through browser caching

The effective use of scanning tools to evaluate the security of a web application requires that the testing team understand the complexities of the target web applications and the importance of properly configured assessment tools. Every engagement requires a thorough understanding of the application, ensuring technologies, authentication and session management mechanisms are thoroughly understood before automated scanning begins. Tools such as IBM Security AppScan[2] and Burp Suite [3] are useful for performing deep manual and automated assessments of web-based applications

Another technique for protecting web servers and applications from malicious attacks is through the use of Web Application Firewalls (WAF). These are firewalls that examine traffic going to and

[2] http://www-03.ibm.com/software/products/en/appscan

[3] https://portswigger.net/burp

from a web application in an effort to detect and block real-time exploits or attack attempts. WAFs also have a much deeper understanding of the application than network Intrusion Detection and Prevention Systems (IDPS) systems. WAFs use a combination of detection techniques including *signature-based*, *rule-based* and *anomaly-based*, and can be configured to either adopt a white-list (default deny all transactions except those known to be valid or trusted) or black-list (default allow, examining transactions for indicators of an attack). Regardless of the detection strategy selected, WAF undergoes a learning period to learn how traffic to and from the web application behaves so that anomalies can be detected.

OWASP Application Security Verification Standard Project

The Open Web Application Security Project (OWASP) is an open organisation that aims to improve the security of software providing information and tools to individuals and organisations for application security. Through this organisation, several open frameworks and tools have been developed providing mechanisms for securing web applications. Security of an application begins at its development and it is therefore recommended to follow the OWASP Developer Guide for secure software engineering when developing applications for the web such as those proposed in the ESCUDO-CLOUD use cases.

OWASP identify ten proactive security techniques that should be included in software development processes [4]:

- Verify for Security Early and Often - Integrate security testing plan into development cycle
- Parametrize Queries - Prevent untrusted inputs from being interpreted as part of an SQL command
- Encode Data - Prevent injection attacks by sanitising characters to a safe form for the target interpretor
- Validate All Inputs - Check that all input is syntactically and semantically valid
- Implement Identity and Authentication Controls - Verify that an individual or entity is who it claims to be
- Implement Appropriate Access Controls - Apply rules to who can access what services or resources
- Protect Data - Encrypt data in transit or at rest
- Implement Logging and Intrusion Detection - Used to identify attacks early and to conduct investigations after the attack
- Leverage Security Frameworks and Libraries - Guard against security related design and implementation flaws by using rigorously tested secure libraries and frameworks
- Error and Exception Handling - Handle all errors and exceptions to ensure application behaves reliably and maintains a stable state

[4] OWASP Top Ten Proactive Controls 2016

While each of these techniques are relevant to all secure development processes, a number of them bear particular relevance to ESCUDO-CLOUD. A publicly exposed API presented to users of ESCUDO-CLOUD should ensure that any parameters passed to the service are sanitised. Implementing this prevents SQL injection attacks that can lead to a compromise of the application database. Potentially more devastating is the ability of an attacker to run OS commands against the OS hosting the database. Most development frameworks support query parametrization to mitigate the risks of such attacks. Closely linked to this is the validation of all inputs to ensure that any data entered by users is checked so that it is both syntactically and semantically valid before being used.

As the use cases selected for the demonstration of ESCUDO-CLOUD require the user management to access features and services, the correct implementation of identity and access controls is required to verify user identities. Session management should form part of this so that servers used to deliver the applications maintain the state of a user interacting with it. Sessions are tracked by a unique session identifier, that is computationally impossible to predict, which can be passed between the client and server when transmitting and receiving requests. In order to reduce the risk of an attack during an active session (e.g., session hijacking), an inactivity timeout for each session should be set. The determination of a timeout period should reflect the sensitivity of the asset which it protects. To provide the authentication component of the access controls, there are a multitude of approaches to select from, including password, smartcard, biometric, token-based authentication etc.

When considering the integrity of the platform and applications, logging and monitoring are invaluable tools to detect intrusions and irregular behaviour quickly which can limit the impact of an attack and help to prevent further attacks. Logging and tracking security events ensures that your cyber security controls and testing are up to date with real world attack campaigns and strategies. The OWASP AppSensor project[5] describes a framework for implementing application level intrusion detection and automated response features into existing applications. Similar to the WAF discussed in Section 3.2.1, this approach is distinct from traditional IDPS that operate at the network level. The core idea of the project is to use detection points in the application to gain visibility into the internal operations of the application. This information can be useful for the advanced detection of unusual activity or behaviour of an application before an attacker has the opportunity to exploit vulnerabilities in the system/application. Building on existing tools and frameworks, AppSensor aims to:

- Detect attackers (not vulnerabilities)
- Act as an application-specific solution
- Avoid the use of signatures for attack predictions
- Allow applications to adapt in real-time to attacks
- Reduce the impact of an attack
- Provide security intelligence

With full knowledge of the application business logic, and the roles and permissions of the users, the AppSensor framework can make informed decisions about irregular patterns in be-

[5] `https://www.owasp.org/index.php/OWASP_AppSensor_Project`

haviour. Once an attack is identified, an alert can be raised to trigger a response action (e.g., shut down application VM, quarantine application, restrict access to application, etc.).

The objective of the OWASP Application Security Verification Standard (ASVS) Project [OWA16] is to provide a standard basis for testing web applications security controls. Adoption of the recommendations listed in the project helps to establish confidence in the security of a web application. The standard defines three security verification levels depending on the sensitivity of the application and data:

- Level 1: All software.
- Level 2: Applications that contain sensitive data, which require protection.
- Level 3: Critical applications - applications that perform high value transactions, contain sensitive data, or require the highest level of trust.

ESCUDO-CLOUD aims to deliver secure services to cloud customers and target use cases are considered to deal with sensitive data. Therefore, guidance for application security should be aligned with the recommendations for ASVS Level 2 and 3. Attackers here are considered to be technically capable and focused on specific targets. ASVS recommends for Level 1 to conduct automated penetration testing to provide as much coverage as possible for the application. At level 2, penetration testing activities will require more detail about the application, including access to documentation and source code. Penetration testing alone is insufficient for Level 3 and should be supported with additional activities such as system configuration review, malicious code review, and threat modelling.

ASVS outlines a series of recommendations categorised across a number of domains including Architecture, Authentication, Session Management, Malicious Controls, Business Logic, etc. Each category identifies controls to implement secure applications according to each of the ASVS levels (listed above). A sample of these controls relevant to the protection of application integrity are listed in Table 3.1. Implementing these controls can increase the security of applications and thereby reduce the risk of a vulnerability being exploited by an attacker.

Control	**ASVS Level** 1	**ASVS Level** 2	**ASVS Level** 3
Architecture, design and threat modelling			
Verify that all application components are identified and are known to be needed	y	y	y
Verify all security controls (including libraries that call external security services) have a centralized implementation			y
Verify the application has a clear separation between the data layer, controller layer and the display layer, such that security decisions can be enforced on trusted systems		y	y
Malicious controls			
Verify all malicious activity is adequately sandboxed, containerized or isolated			y

Verify that the application source code, and as many third party libraries as possible, does not contain back doors, Easter eggs, and logic flaws in authentication, access control, input validation, and the business logic of high value transactions			y
Business logic			
Verify the application will only process business logic flows in sequential step order, with all steps being processed in realistic human time, and not process out of order, skipped steps, process steps from another user, or too quickly submitted transactions		y	y
Files and resources			
Verify that the web or application server is configured by default to deny access to remote resources or systems outside the web or application server		y	y
Verify the application code does not execute uploaded data obtained from untrusted sources	y	y	y
Configuration			
All components should be up to date with proper security configuration(s) and version(s)	y	y	y
Verify application deployments are adequately sandboxed, containerized or isolated to delay and deter attackers from attacking other applications		y	y
Verify that the application build and deployment processes are performed in a secure fashion		y	y
Verify that all application components are signed			y
Verify that third party components come from trusted repositories			y

Table 3.1: Selection of Relevant ASVS Controls

3.2.2 File Integrity Monitoring

File Integrity Monitoring (FIM) is a service that validates the integrity of OS and application files by verifying the current state of the files against a known trusted baseline measurement. To reduce the computational overhead of comparing large files, the method typically uses the hash or cryptographic checksum of the files for the comparison. The approach is useful for the detection of changes to files, configuration values, credentials, security settings and policies and data. The implementation of a FIM process is a requirement in multiple standards including PCI-DSS [PCI16a], HIPAA [NIS08] and SANS Critical Security Controls. If not implemented correctly and the output of the service monitored, an attacker could add, remove, or alter configuration file contents, the OS, or application executables. If left undetected, the attacker could bypass existing security controls giving them freedom to manipulate the application environment to achieve their malicious objective.

The most basic method for comparing files is to first generate the hash of the file. Following security best practices, the file and hash should never be transmitted together. In order to verify the state of the files remained unchanged and have not been tampered with, the hash is recalculated

and compared against the original baseline hash value. SHA-256 is the current recommended cryptographic algorithm to use when generating hashes. OpenSSL, the popular open cryptography and SSL/TLS toolkit provides an easy way to integrate library of cryptographic algorithms including hash functions such as SHA-256 and MD5. OpenSSL provides a command line interface to generate the hash of a file and has a C API for inclusion in application development. For Java applications, the Java SE security platform provides a comprehensive set of cryptographic features.

There are a number of commercial offerings available in the market to manage integrity monitoring for files and OSs, including Intel Security Change Control[6], Cloud Passage Halo[7], AlienVault PCI-DSS File Integrity Monitor[8] and Tripwire File Integrity Manager[9]. These products aim to address compliance issues within highly regulated industries moving their services to cloud infrastructures, continuously tracking changes to files, registry keys and security settings. Logging is another key offered featured that can be important for detecting indicators that your systems have been compromised. From an architectural perspective, these tools require a lightweight agent to be installed on the host and managed from a central server.

3.3 Trusted Computing

This section introduces the concepts and technologies available to establish a root of trust that provide the foundation for trust at higher levels (e.g., OS, application). These technologies are essential in order to provide some assurance over the integrity of the execution environment. First, vendor specific (VMWare) controls will be highlighted to illustrate the configuration options available to monitor virtual environments for unauthorised modifications.

To achieve better guarantees over confidentiality and integrity, the Trusted Computing Group (TCG) [10] devised a set of hardware and software technologies to enable the realisation of a trusted platform. The TCG concluded that software based security solutions are not enough to deploy a truly trusted platform. This section will discuss some of the technologies available to establish a trusted compute platform of cloud computing. A core technology for the implementation of a trusted environment is the Trusted Platform Module (TPM) which will be discussed, along with the various approaches for OS and application verification that have been built on the foundations of TPM.

3.3.1 Virtual Machine Integrity

Despite the well documented benefits of cloud computing, there is a fundamental conflict between the usage of public infrastructure to host data that is inherently private. Outsourcing compute and storage resources introduces a layer of obfuscation between the data owner and the data itself. Visibility and control over the data is lost. The first component to be addressed in providing assurance of the environment is the virtual machine. Providing assurance at this point reduces the possibility of software bugs, hardware failures, misconfiguration, or malicious attacks that affect the security or behaviour of the environment. Without any transparency of how the VM or CSP are configured and how operations are performed, the potential exists for a CSP to operate in a

[6] http://www.mcafee.com/us/products/change-control.aspx
[7] https://www.cloudpassage.com/solutions/compromise-detection/
[8] https://www.alienvault.com/solutions/pci-dss-file-integrity-monitoring
[9] http://www.tripwire.com/it-security-software/scm/file-integrity-monitoring/
[10] http://www.trustedcomputinggroup.org

way that might affect the confidentiality or integrity of the data. For example, a cloud service may be configured to provide an expected level of encryption for end user data, however, under periods of heavy load the CSP may apply weaker security as a temporary measure to maintain the service. Though data may only be protected with weaker mechanisms for a short period of time, it opens a window for attackers to exploit.

One novel approach to verifying the computational integrity, based on the idea of uncheatable distributed computing [GM01], is to perform the same computations on multiple CSPs and compare the results. This approach however is more suitable for static processes that are applied to sets of data and have easily measured results. The processes outlined by the use cases in ESCUDO-CLOUD require user interaction which is harder to quantify. Additionally, the overhead incurred by replicating the storage and access requests of large volumes of data would hinder the performance of the service. This approach also potentially opens the attack surface which exposes the end user data to further vulnerabilities.

A similar approach, in the context of Use Case 4, would be to split encrypted files on the client side into multiple blocks and forward each piece to a separate CSP running the ESCUDO-CLOUD service in parallel. In this way, no single CSP has the entire file and subsequently, in the event that a key becomes compromised, does not have access to the protected data. Since data in this use case is encrypted on the client side, the primary risk is to the key management server. Following recommended security guidelines, this server should reside on a separate physical server to the ESCUDO-CLOUD middleware.

Typically, hypervisor platforms such as VMware's ESX/ESXi host platform provide mechanisms for the verification of the VM integrity. To understand how VM file integrity is performed, we first introduce in Table 3.2 the various files that make up the VM running on an ESX/ESXi host.

File	Description
VMX	The primary configuration file of a VM. Every aspect of your VM is detailed in the VMX file, and any virtual hardware assigned to your virtual machine is present here.
VMXF	A supplemental configuration file for virtual machines.
-flat.VMDK	Stores the content of the VM's actual hard disk drives.
VMSD	A centralized file for storing information and metadata about a snapshots of a VM.
NVRAM	Contains the BIOS of the VM.

Table 3.2: Virtual Machine Files

These are critical files in the ESX/ESXi file system that should be monitored for changes, accidental deletion or corruption. By creating a catalogue of file permissions and hashes representing

file state, administrators can monitor and maintain these files to ensure they aren't accessed or modified without permission. VMware recommend to *establish and maintain file system integrity* [11]. While most configuration files are controlled via an API, a set of specific configuration files, used to govern host behaviour, are exposed with the vSphere HTTPS-based file transfer API. Tampering with these files has the potential to enable unauthorized access to the host configuration and VM. A number of methods are available (Managed Object Browser, vCLI, etc.) to monitor these files and their contents, ensuring that they have not been maliciously or accidentally modified. Similar monitoring mechanisms should be used according to the host platform recommendations.

Consideration should also be given to the installation of OSs and applications. It is recommended to always check the hash after downloading an ISO, installation bundle or patch to ensure the integrity and authenticity of the downloaded files. After downloading media, use the MD5 sum value to verify the integrity of the download. Compare the MD5 sum output with the value posted on the distributors website. ESXi includes an additional layer of security, using digital signatures to ensure the integrity and authenticity of modules, drivers and applications as they are being loaded by the VMkernel. In this, ESXi can identify the providers of modules, drivers or applications and whether they are *VMwareCertified*. VMwareCertified is one of the four vSphere Installation Bundle (VIB) [12] acceptance levels supported by the ESXi image profiles [13].

1. VMwareCertified - VIBs created, tested and signed by VMware.
2. VMwareAccepted - VIBs created by a VMware partner but tested and signed by VMware.
3. PartnerSupported - VIBs created, tested and signed by a certified VMware partner.
4. CommunitySupported - VIBs that have not been tested by VMware or a VMware partner.

With the exception of CommunitySupported, VIBs meeting the other acceptance levels can be installed on ESXi hosts. CommunitySupported VIBs are not supported and do not have a digital signature.

3.3.2 Container Integrity

VMs are no longer the only popular virtualisation solution for the deployment of applications and services. Container technologies have risen in prominence in recent years, in particular Docker, as a way to deploy lightweight, flexible services on a virtualized infrastructure. The rapid uptake in the technology left a number of security vulnerabilities exposed which have been quickly addressed in successive releases. The current version of Docker has well documented security best practices and has addressed most significant security concerns.

In Docker, images containing OSs, applications and/or services, are stored in public (or private) registries (a stateless, highly scalable server side application that stores and distributes Docker images) and are pulled using Docker Engine. This model for image distribution raises concerns over the integrity of the images and of the publisher of data received from the registry. Digital signature verification is a relatively new feature, introduced via the *Docker Content Trust* feature released in Docker 1.8.

[11] VMware vSphere 5.1 Documentation Center (`http://pubs.vmware.com/vsphere-51/index.jsp`)
[12] A collection of files packaged into a single archive to facilitate distribution
[13] VMware vSphere 6.0 Documentation Center (`http://pubs.vmware.com/vsphere-60/index.jsp`)

Content Trust integrates the Update Framework (TUF)[14] into Docker using Notary[15]. TUF is a comprehensive, flexible framework that helps developers to secure new or existing software update systems (software installation application running on a client) that might be vulnerable to attack. Notary is a Docker project that builds on TUF and consists of a server and a client for running trusted collections, making it easier to publish and verify content.

Content Trust builds trust into the system by allowing operations within a remote Docker registry to enforce client-side signing and verification of the image. Docker also implements digital signatures on data sent to/from the remote Docker registries, allowing the client to verify the integrity of the image and verify its publisher. This establishes an environment in which image publishers can sign their images and image consumers can verify those signatures. When a publisher uploads an image to the registry, Docker Engine signs the image locally with the publishers private key. The corresponding public key can then be used to verify the image when it is requested to ensure that it has not been corrupted or tampered with.

Content Trust has two distinct keys, a *Root* key and a *Tagging* key that are generated the first time a publisher pushes an image to the registry. The Root key is the root of trust for an image tag. A user only has one Root key and it is stored offline (also known as the Offline key). There exists a unique Tagging key for every repository. When a user first runs a pull command, they establish trust to the repository using their Root key.

While this section and Section 3.3.1 have examined the vendor specific tools available through VMware to manage VM integrity, the tools themselves are software applications that are potentially vulnerable to compromise. Therefore it is necessary to drill deeper into the chain of trust to establish a so called *root of trust* from which the integrity of each subsequent layer is based. Section 3.3.3 will explore the concept of the Trusted Platform Module (TPM) that can be used to provide such a root of trust.

3.3.3 Trusted Platform Module

One of the key standards developed by the TCG was the Trusted Platform Module (TPM) [16]. It is used to assist the remote attestation of the platform, that basically measures and record the configuration information of the core components (BIOS, firmware, OS, software) of the platform. This information is stored in the TPM which acts as a root of trust for the platform.

Its implementation is available as a chip that is now bundled with commodity hardware. It provides cryptographic operations such as key generation, encryption/decryption, signing/verification, hashing, random number generation and migration of key between TPMs. TPMs are resistant to hardware and software attacks and contain an *endorsement private key (EK)* that uniquely identifies the TPM (and therefore the physical host). Manufacturers of TPMs sign the corresponding public key and provide a certificate to validate the key and ensure its correctness. It also provides secure storage for small amounts of information such as secret keys.

TPMs allow a system to gather and attest system state, store and generate cryptographic data and prove platform identity. Remote attestation involves the creation of a hash key based on the hardware and software configuration of the system. This hash value can be compared with a measured hash value of the current state of the system to verify the integrity of the system including the OS and applications. Figure 3.1 details the architecture of a TPM chip, identifying the

[14] `https://theupdateframework.github.io`
[15] `https://github.com/docker/notary`
[16] `http://www.trustedcomputinggroup.org/tpm-main-specification/`

core cryptographic, storage and processing components which are sealed inside tamper resistant packaging.

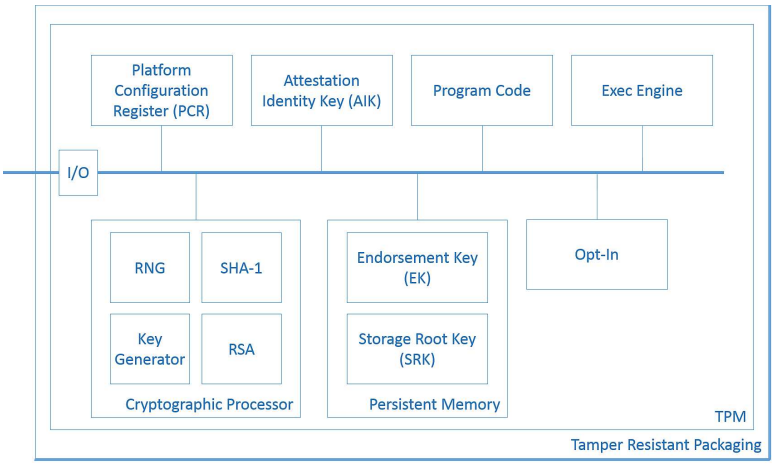

Figure 3.1: TPM Architecture

The design of the cryptographic processor implemented on the TPM focuses on delivering specific cryptographic functions in order to provide a cost effective solution. For example, the TPM does not provide an implementation for AES or any symmetric key algorithm, though the TPM can store symmetric keys. Instead, only the asymmetric RSA algorithm is implemented and is used for key generation, digital signatures and encryption/decryption of keys. The original specification of TPM supported only SHA-1, though SHA-256 is now supported in the current TPM standard. Finally, the TPM cryptographic processor contains a random number generator (RNG) that is used to protect against replay attacks and to generate random keys.

Attestation identity keys (AIKs) created by the TPM are related to the EK. They are linked to the local platform through a special certificate for the AIK that is created and signed by a CA. The AIKs are used to digitally sign internal TPM data in order to provide attestation (provides proof of data known by the TPM). The validity of the integrity measurements and the AIK can be determined by a verifier. To provide attestation of the platform, a set of Platform Configuration Registers (PCR) are digitally signed using the AIK.

The TPM contains several PCRs that can be used to securely store the digests of previously recorded configurations that can be used to detect changes in the system. Products such as Microsoft BitLocker and Intel Security Endpoint Encryption use these registers to store configuration and secret key material to enhance the security of their system integrity and disk encryption services.

As it can be seen in Figure 3.1, the EK (introduced above) and Storage Root Key (SRK) are stored in persistent memory. The SRK is the root key in the key hierarchy of the TPM. Every key that is generated by the TPM has its private component protected with its parent key, creating a chain to the SRK. Of the remaining components of the TPM: the program code contains the firmware for the measuring platform components and is also known as the Core Root of Trust for Measurement (CRTM); the program code is executed in the execution engine; and the Opt-In component contains mechanisms for the management of the TPM modules state.

Trusted platforms leverage TPM capabilities to enable remote attestation. On booting the host computes a measured list, *ML*, consisting of a sequence of hashes of the software involved in the boot sequence (BIOS/UEFI, bootloader, software implementing the platform, etc.). This

list of hashes is stored securely in the TPM. To attest to the platform, the user challenges the platform running at the host with a nonce, *n*. The TPM then creates a message containing *ML* and *n*, encrypted with it's EK. This message is sent to the user who decrypts the message with the corresponding public key, authenticating the host. Provided that the nonces match and the list of hashes corresponds to a trusted configuration, the user can determine the trustworthiness of the host. This provides a basis for determining the integrity of all software running on the platform, from the hypervisor itself to all OSs and applications running inside VMs.

VMM

A VMM, also known as the hypervisor, is what enables multiple OSs to run on shared physical resources. It is a software layer between the OS and the hardware, providing the abstraction that aggregates physical resources (compute, storage, network) and presents virtual resources. The VMM also provides isolation between VMs on the same host. Typically, during the boot process of a VMM, an initial *management* VM is started that is necessary for starting further VMs. The VMM guarantees its own integrity until the machine reboots. Therefore, a remote party can attest to the platform running at the host to verify that a trusted VMM implementation is running, and make sure that the computation running in the VM is secure.

While VMMs are mature technologies with multiple vendors that are capable of abstracting physical resources into virtual resources, indistinguishable by the OS from the physical components, TPMs provide a new challenge in this abstraction process. The root of the issue is the introduction of new states of operation for the VM allowing the VMM to suspend the system and resume it later. VM migration is also a challenge for integrating a TPM into a security solution, as the TPM cannot physically move with the migrated VM. The following section will introduce the concept of the virtual TPM (vTPM) and discuss how these challenges can be addressed to provide TPM functionality to protect the integrity of images, OSs and applications.

A Trusted VMM (TVMM) builds on the properties of a traditional VMM to provide additional capabilities that provide guarantees about the security and integrity of the VMs. VMMs already provide isolation to applications running in different VMs which is essential to maintain confidentiality and integrity. VMMs too are designed to be secure. Without the security challenges of traditional OSs dealing with file systems, networks, etc. the VMM is only concerned with relatively simple abstractions of resources. The proposed Terra TVMM builds on this foundation by providing additional capabiliies:

- Root Secure: Ensures that even the CSP administrator cannot break the privacy and isolation guarantees the TVMM provides to VMs.
- Attestation: Allows an application running on a VM to identify itself securely to a remote party, establishing trust in the application for the remote party.
- Trusted Path: Provides a trusted path between the user to the VM/application so that each endpoint is aware of who they are interacting with. Also ensures the confidentiality and integrity of the communications between the end points.

vTPM

VMMs and hypervisors are naturally good at isolating workloads from each other because they mediate all access to physical resources by VMs. A root of trust established in hardware (such as

the TPM), provides resistance to software attacks and provides a mechanism to verify the integrity of all software and VM images running on a platform. It is obvious however that TPMs were not designed to be accessed by multiple systems at the same time. Virtualizing the TPM extends the capabilities of the TPM, making them available to all VMs running on a platform. In this way, each VM is provided with its own private TPM. As it is done with the abstraction and distribution of physical resources (compute, storage, network) to support multiple VMs, a single hardware TPM can be virtualized to provide multiple virtual TPM instances that can carry out the functionality of the hardware TPM.

The implementation of a vTPM has a number of requirements as follows:

- Consistent usage model: A vTPM must provide the same usage model and TPM command set to VMs as it is provided by a hardware TPM to physical machines
- vTPM - VM binding: The association between a VM and its vTPM must be maintained for the duration of the VM lifecycle even if the VM is migrated to another host
- vTPM - TPM binding: The association between a vTPM and its TPM, on which its functionality is derived, must be maintained
- It must be possible to distinguish between a vTPM and a TPM

A practical and secure implementation of a vTPM presents some challenges [PSD+06] though implementations are available from different vendors. The key security concern when creating an instance of a vTPM, is the establishment of a chain of trust from the physical TPM to each vTPM. This is achieved via the management of signing keys and certificates. Despite the measures enforced to extend the chain of trust, some applications and OSs relying on TPM functionality should be made aware that they are using a vTPM so that the correct procedures and additional steps can be implemented.

A secondary challenge with vTPMs relates to how to deal with the migration of vTPM instances between hosts when the associated VM migrates. With the EK tightly coupled with to the hardware configuration of the host, the migration of a VM would affect the ability to validate digital signatures from the TPM as the underlying hardware and software is liable to change. Therefore, it is critical that the trust established in the initial environment is carried over to the vTPM environment. Secret key data stored and protected in the vTPM instance must be secured in transit to an adequate level ensuring the persistence of security levels offered by the vTPM.

Most developments for vTPM implementations have been in the Xen platform [PSD+06]. Xen provides documentation for the configuration of a platform that uses virtual TPMs to provide TPM security functions to guests running on the platform [17]. The document itself notes that there are *tradeoffs between flexibility and trust which must be considered when implementing a platform containing vTPMs*. Two examples are provided that present configurations for a Trusted *Domain 0* (simple, flexible configuration) and for a domain builder with static vTPMs. The most notable restriction of the latter example is that it is not possible to create additional vTPMs once the host is booted. In addition, VMs with associated vTPMs cannot be rebooted without rebooting the entire host.

IBM proposed two solutions for the implementation of a vTPM for Xen that met the security and performance requirements for critical systems involving highly sensitive operations [18].

[17] `http://xenbits.xen.org/docs/unstable/misc/vtpm-platforms.txtAccessedAugust2016`
[18] `http://researcher.watson.ibm.com/researcher/view_group.php?id=2850`

The first solution is to host the vTPM functionality on the IBM PCI-X Cryptographic Processor (PCIXCC) which provides tamper resistance ensuring protection of private keys from attackers. The architecture for this solution is presented in Figure 3.2 showing the PCIXCC subsystem running TPM functions for multiple vTPMs. A *DOM-TPM* introduced here acts as a back-end (TPM BE) proxy for the PCIXCC vTPM, making TPM instances available to all other VMs, running a TPM front-end (TPM FE), on the host.

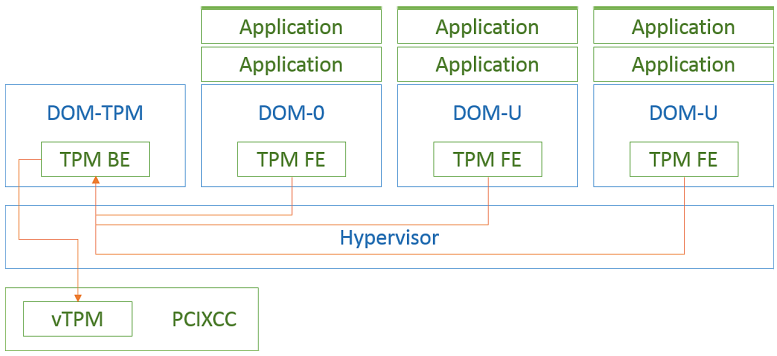

Figure 3.2: vTPM implemented using PCIXCC

The other solution proposed by IBM is illustrated in Figure 3.3 and uses a software implementation of TPMs to provide TPM functionality. The DOM-TPM domain is again associated with the physical TPM and is used to provide the software TPMs with their functionality. Research into this approach for implementing a vTPM on Xen was extended, with detailed instructions on how to setup a "mini-os vTPM subsystem" available at `http://xenbits.xen.org/docs/unstable/misc/vtpm.txt`.

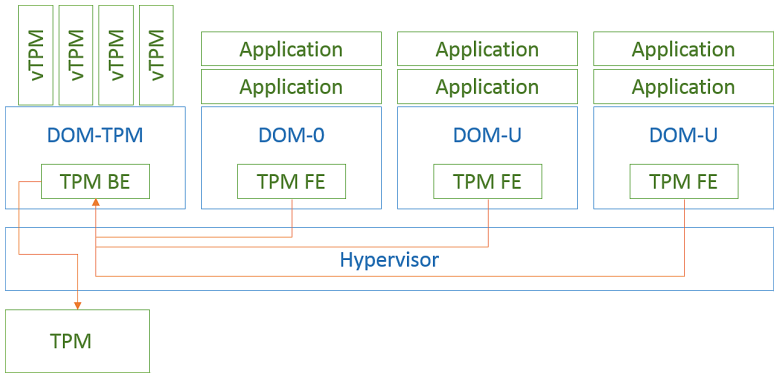

Figure 3.3: vTPM implemented using TPM on motherboard

These advances in finding practical and secure solutions to implement vTPMs open the possibility of providing cost effective TPM functionality to VMs which will enable users to attest to the integrity of their VMs, images and applications before booting. This root of trust in their environment will give cloud service users confidence in the state of their operations.

ESXi Integration

ESXi can use TXT/TPM to verify that the booted kernel and some of its respective loaded modules have not been unexpectedly modified through an unauthorized update or some other malicious type of change. This capability is enabled by default in ESXi and cannot be disabled. However, to take

advantage of this security feature, it should be verified that TXT/TPM is also enabled in BIOS.

While an event is logged for TXT/TPM, there is no user interface to view the TXT/TPM measurements that are made of the kernel and the respective loaded modules within the vSphere GUI. 3rd party solutions can use an API call to verify that the kernel and those modules that are inspected in the implementation of TXT/TPM by VMware have not been modified.

ESXi provides additional VMkernel protection with the following features:

- Memory Hardening: The ESXi kernel, user-mode applications, and executable components such as drivers and libraries are located at random, non-predictable memory addresses. Combined with the non-executable memory protections made available by microprocessors, this provides protection that makes it difficult for malicious code to use memory exploits to take advantage of vulnerabilities.

- Kernel Module Integrity: Digital signing ensures the integrity and authenticity of modules, drivers and applications as they are loaded by the VMkernel. Module signing allows ESXi to identify the providers of modules, drivers, or applications and whether they are VMware-certified.

- TPM: Each time ESXi boots, it measures the VMkernel and a subset of the loaded modules (VIBs) and stores the measurements into PCR 20 of the TPM. This feature is enabled by default and cannot be disabled.

3.3.4 Trusted Cloud Computing Platform (TCCP)

The concept of a TCCP enables cloud IaaS providers (e.g., Amazon EC2) to provide a closed environment guaranteeing the confidentiality and integrity of operations and data on VMs executed inside it. Early implementations of a TCCP, such as Terra [GPC+03], enables cloud customers to prevent the CSP from inspecting or interfering with VM operations. This system also provides a remote attestation capability that allows the customer to determine, in advance, whether or not the CSP can securely run their application. This approach was suitable for VMs running on a single host. However, the concept of federated cloud and elastic cloud, along with the growth in CSPs is moving computations, data and VMs beyond a single host environment into datacentres comprising of thousands of hosts that may be spread across a wide geographical area. Compute resources are allocated dynamically and customers have little visibility of how or where their data is processed.

A TCCP proposed by Santos et. al. [SGR09] combines trusted computing and secure hypervisors to provide a secure platform in an environment where compute resources are outsourced to multiple hosts, guaranteeing that no CSPs administrative staff can inspect or interfere with customer operations or data. In addition, users of VMs in the environment can attest to the provider and determine the security status of their VMs. An illustration of this TCCP design is provided in Figure 3.4. The components of the TCCP include a set of trusted nodes (N), the trusted coordinator (TC) and an untrusted cloud manager (CM). In this scenario, the VM is launched on N, and the CM makes a set of services available to users. The TC, hosted and managed by a 3rd party external trusted entity (ETE), is used to handle all attestations.

Each node, N, runs a *trusted VMM* (TVMM) and prevents privileged users from inspecting or interfering with them. To provide maximum security and integrity, nodes use a TPM to support a secure boot process to install the TVMM. The TC maintains a record of the nodes located in the security perimeter and attests to the nodes platform that it is running a TVMM. In order to

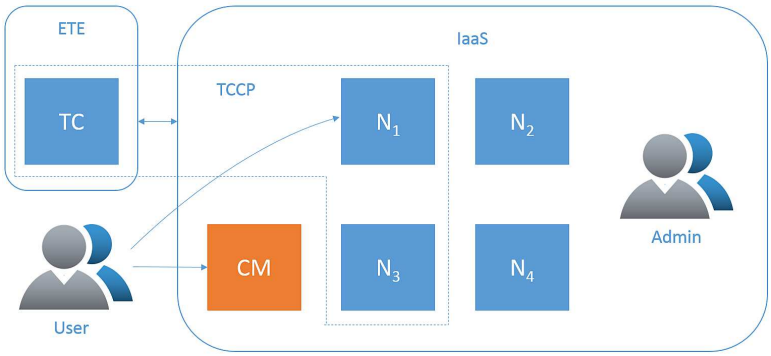

Figure 3.4: Trusted Cloud Computing Platform

ensure the integrity of VMs in the TCCP, the TC confines the execution of a VM to a trusted node. In addition, during transit, the TC prevents the inspection or modification of the VM state. The TCCP specifies several protocols to address the various states of the VM (launch, suspend, resume, migrate) to ensure at all points that VMs are only run within the trusted perimeter and are only accessible to authorised users.

Referring back to the typical insider attack description provided at the beginning of this chapter, a malicious CSP staff member requires administrator permissions on the target VMs in order to inspect or interfere with its applications. While each vendor will enforce different security mechanisms to prevent such an intrusion, here we assume the attacker has the technical ability and appropriate permissions to manipulate the physical host on which the VM is run and to remotely login to the target VM with root privileges. Under these conditions, the TCCP must be able to restrict the movement of VM compute resources and restrict the capabilities of users with root privileges to prevent them accessing memory. As the TC is hosted by the external ETE, CSP administrative staff have no privileges inside the ETE and therefore cannot tamper with the TC. In this way, even a skilled insider with root privileges will be unable to inspect or modify the state of a VM running inside a trusted node. The only significant drawback to this approach, as with any service relying on an independent 3rd party, is that the system depends on the TC and ETE to be available for the service to run.

3.3.5 Direct Anonymous Attestation

Direct Anonymous Attestation [BCC04] was a scheme developed by the Trusted Computing Group [19] as an approach to remotely authenticate a TPM (introduced in Section 3.3.3). DAA involves several zero-knowledge proofs to guarantee the trustworthiness and privacy of an appropriate platform. In the context of providing assurance over the state of an application and VM, DAA provides a mechanism for one machine to attest its configuration state to another machine/user.

The use case addressed by DAA is as follows: Consider a TPM integrated into a host platform. A user of the platform is requested by a *verifier* to provide assurance that the user is indeed using a platform that contains a TPM. In other words, the verifier wants the TPM to authenticate itself. There is an additional requirement that the user does not want to disclose their identity to the verifier and therefore the verifier can only know that the user uses a TPM, but not which specific one. In the event that the verifier gains knowledge of the users associated TPM, the verifier would

[19] `http://www.trustedcomputinggroup.org`

be able to track all transactions with that user.

The solution devised by TCG assumes a 3rd party CA that has knowledge of the public component of all TPM EKs. In order to authenticate itself to a verifier, a TPM generates a second RSA key pair, the AIK. The public component of the AIK is signed using the EK and is sent to the CA who checks its validity. The CA issues a certificate for the AIK which can be presented to the verifier. In this way, a new AIK can be generated by the TPM for every transaction with the verifier without exposing its EK to the verifier and therefore protecting the identity of the user.

Similar schemes using ECC in place of RSA have been developed [BCL08][Che09][CPS10], taking advantage of shorter key and signature lengths and reduced computational load on the TPM. A set of tools for DAA are available from IBM [20] which can be used to verify the DAA commands on a TPM implementation. According to a 2014 ENISA report, adoption of this technology has been limited.

SGX

SGX[21] is a relatively new technology emerged from Intel that provides a set of new CPU instructions that can be used by applications to establish secure regions of code and data. The technology aims to help application developers to protect code and data from being viewed or altered. The central challenge is the execution of software on a remote computer that is owned and maintained by an untrusted 3rd party such as a CSP. SGX considers the possibility that all privileged software and users are potentially malicious and therefore need to be restricted from accessing sensitive operations. Figure 3.5 illustrates the basic problem with trust in computing. Protected Mode (rings) protects the OS from applications and applications from each as it is the situation with the application on the left side of the figure. However, if a malicious application is loaded that exploits a security vulnerability to gain full access privileges, it can then tamper with the OS and other applications.

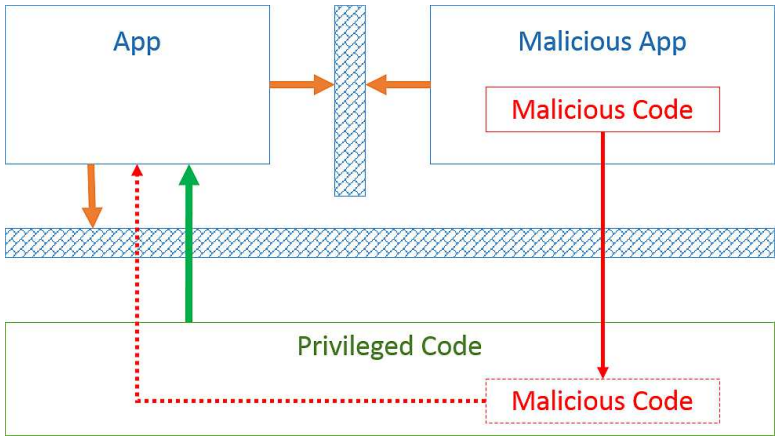

Figure 3.5: Trust in Computing

The core idea of SGX is the *enclave*, which are protected areas of execution. Application code can be placed into an enclave using the APIs and tools provided by the SGX SDK. SGX builds on the efforts towards establishing a trusted computing model and relies on software attestation to prove to the user that they are communicating with a specific piece of software running on

[20] https://www.zurich.ibm.com/security/daa/IBM-DAA-TPM-TestSuite-Overview.html
[21] https://software.intel.com/en-us/sgx

a secure container hosted by trusted hardware. Once the system state is attested, the user can operate within the secure enclave which is protected from the OS upon which it is running. Figure 3.6 illustrates the process and architecture for executing sensitive operations of an application in a trusted enclave.

At runtime, the SGX instructions build and execute the enclave in a special protected memory region (available on Intel Skylake chips) that have restricted access points defined by the application developer. The first step in the development of an application using SGX is to construct trusted and untrusted parts of the application (1). The application runs and creates the enclave which is placed in trusted memory (2). A trusted function *Trusted()* is called, at which point execution of the application is transferred to the enclave (3). While the enclave can see all data and processes, no external process or user can gain access to this portion of memory (4), thus ensuring the confidentiality and integrity of the operations. Finally, the trusted function returns to the caller in the untrusted zone without leaking any enclave data from trusted memory (5) and the application continues to its next operation (6).

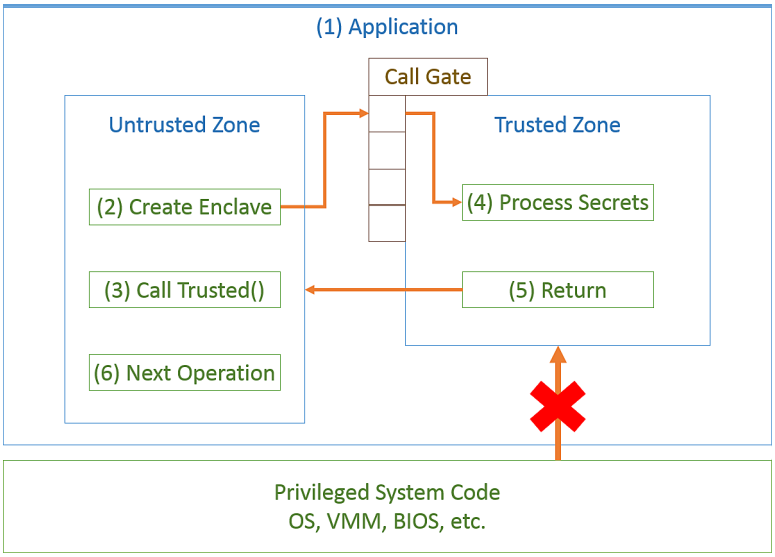

Figure 3.6: Intel SGX

SGX opens up the potential for a wide array of use cases where sensitive operations and operations on sensitive data can be isolated to trusted zones of memory, built upon a hardware root of trust such as TPM. Potential use cases include:

- Guarding applications & data
- Hardening end-point security
- Protecting communications
- Protecting sensitive/confidential data
- Secure analytics workloads
- Secure IoT edge devices and cloud communications

3.3.6 Summary

This chapter has covered a wide variety of techniques for ensuring the integrity of your physical infrastructure, virtual environment, OS and applications. Taking this *ground up* approach to assuring the integrity of applications running in the Cloud (or indeed in any environment) provides users with a traceable root of trust built into the hardware. The core trust anchors in the chain of trust to the application include:

- Trusted Platform - Recent trends in technology, such as TPM and SGX, aiming to build trust into the hardware have opened up opportunities for a host of new application and OS security use cases. Prior to these advancements, mechanisms to protect software vulnerable to attacks at the hardware, firmware or OS level. By ingraining trust into the hardware, subsequent software layers can provide assurances over its state.

- Virtualisation - Virtualisation technologies such as virtual machines and, more recently, containers have revolutionised the way organisations do business, providing a flexible, scalable and affordable platform (through Cloud computing) to acquire compute and storage resources. However, virtualisation introduces another attack vector which needs to be addressed. Technology vendors (e.g., VMWare, Docker) provide comprehensive security guides and recommendations for securing your environment and ensuring it is operating in a known, stable state.

- Application - The final trust anchor is with the application itself. Techniques for analysing and monitoring applications during their development and during runtime were presented. A number of techniques and recommendations (including OWASP recommendations) for securing web applications were also covered that help to reduce the risk of an application becoming compromised.

In addition, processes for assuring the integrity of code and installation binaries during the development cycle of an application were examined. Best practice guidelines for secure development, in particular around the use of external libraries were outlined. Code signing, as well as binary signing, were highlighted as an effective mechanism for distributing trusted content and applications to end users.

The techniques and technologies explored in this Chapter provide a solid set of recommendations for the deployment of trusted applications in trusted environments. It is important for the four use cases to give consideration to these recommendations in deployment of their architectures. In addition, software libraries resulting from the other WP activities should take steps to integrate code signing into their development processes and should make use of digital signatures to ensure the validity of the resulting libraries and binaries.

4. Applicability of Security Testing Techniques to Use Cases

ESCUDO-CLOUD is predominantly driven by the enumeration of requirements from actual Use Cases (UCs) with the resultant articulation of user data ownership scenarios. Thus, ESCUDO-CLOUD explores security testing at the requirements level. Specifically, the security testing is performed in order to mitigate the threats that can potentially violate the user's data ownership requirements of security, functionality and performance. The UCs also form the basis for testing and validation. Each Use Case addresses a specific application scenario, which can be classified as Infrastructure Provisioning, Cloud Storage, and Cloud Processing scenarios.
Use Case 1 (UC 1): This Use Case relates to cloud-storage platform, which supports server-side encryption with flexible key-management solutions, to be used with OpenStack framework. This facilitates user data security in the OpenStack framework
Use Case 2 (UC 2): This Use Case covers secure enterprise data management in the cloud, specifically secure data sharing among the business parties. The goal of this Use Case is to provide solutions that allow a data owner to share information without losing control over his data.
Use Case 3 (UC 3): This Use Case considers the application of data protection as a service via a cloud service store that enables customers to protect their data stored on multi-cloud environments including federated secure cloud storage.
Use Case 4 (UC 4): This Use Case considers cloud service brokers or intermediaries offering a secure cloud data storage capability to their customers while possibly leveraging other cloud providers for storing this data and ensuring that data is protected from such other cloud providers and other users.

The process followed in this section takes the UCs defined in WP1 and the requirements elicited in WP2 (W2.3) to suggest the security testing techniques (described in Chapter 2) applicable to each of them. Figure 4.1 provides an overview of the process followed where requirements elicited from the UCs (as in W2.3) are analyzed covering the aspects of:

- **Applicability**. Certain testing techniques are more suitable than others, depending on the asset to protect (access control infrastructure, databases, etc). For instance, a perturbation analysis would be suitable to check the level of protection of a key based access control, by checking the correct behavior of the system when a malformed or malicious key is used.

- **Threats Impact**. This includes the evaluation of the impact of unfulfilling the requirement over the elements of the system that are involved. For example, requirements related to confidentiality are defined with the objective of protecting databases or servers. This would involve the evaluation of potential threats (according to the software used in the server or database) that would require to apply vulnerability testing techniques.

- **Availability of assets**. The availability, or lack thereof, of certain parts of the system limits the applicability of certain security testing techniques. For example, without access to the code base it is not possible to perform white box testing while without the availability of interfaces it is not possible to perform black box tests.

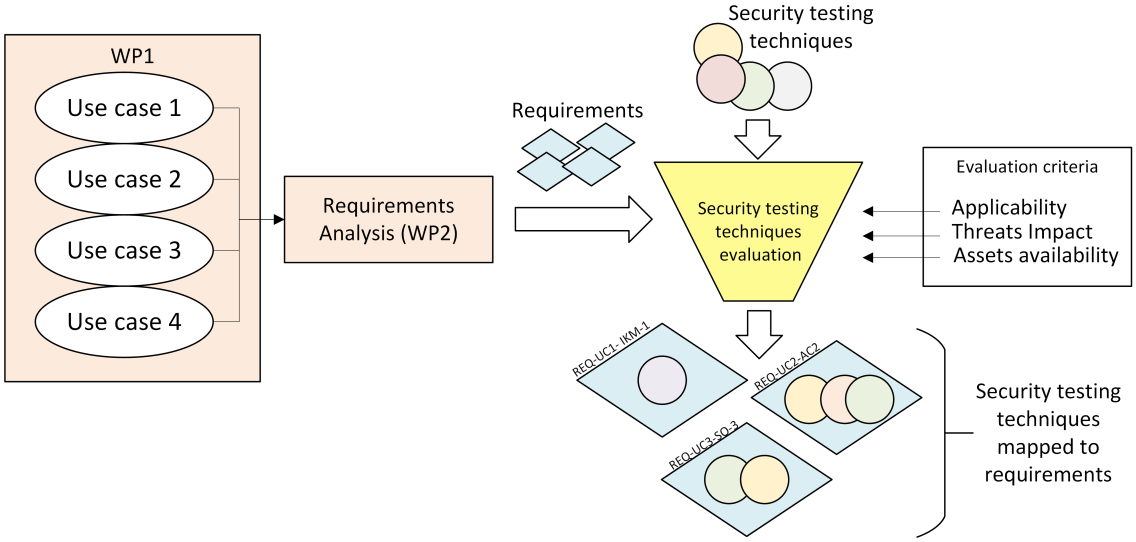

Figure 4.1: Process to evaluate requirements vs security testing techniques

The result of the analysis is a list of potential security testing techniques available to check the fulfilment of the requirements for every UC. For each UC, we identify the security requirements from multiple dimensions as depicted in Tables 4.1, 4.2, 4.3 and 4.4. These tables outline (a) security properties (Confidentiality, Integrity and Availability); (b) sharing requirements; (c) access requirements; (d) required security tests. For each UC requirement, we advocate the suitable security testing technique(s) according to the security test techniques specified in Chapter 2.

Requirement Reference	Requirement Description	Direct Assumptions	Indirect Assumptions	Violation Likelihood (1/Low-10/High)	Assumed Impact	Threat Severity (1/Low-10/High)	Advocated Security Testing Technique	Comments on Applicability of Security Testing
REQ-UC1-IKM-1	CRUD operations for infrastructure keys	Trusted administrator	Key existence	7	Confidentiality violation for tenant data	10	-Vulnerability assessment -Penetration testing	Databases protecting tenant data are subject to threats and vulnerabilities (depending on the technology used). Tests related to the potential vulnerabilities are required to ensure the confidentiality of the data and the validity of the keys used.
REQ-UC1-IKM-1	CRUD operations for infrastructure keys	Secure storage for keys		3	Confidentiality violation for tenant data	10	-Vulnerability assessment -Penetration testing	Databases protecting tenant data are subject to threats and vulnerabilities (depending on the technology used). Tests related to the potential vulnerabilities are required to ensure the confidentiality of the data and the validity of the keys used.
REQ-UC1-IKM-1	CRUD operations for infrastructure keys	Key length sufficient		3	Confidentiality violation for tenant data	10	-Vulnerability assessment -Penetration testing	Databases protecting tenant data are subject to threats and vulnerabilities (depending on the technology used). Tests related to the potential vulnerabilities are required to ensure the confidentiality of the data and the validity of the keys used.
REQ-UC1-IKM-2	Policy-driven and automated infrastructure-key management	Policy language complete	Policy engine functional, automation present	4	Policies leak information about infrastructure keys, expose infrastructure keys	8	-Fuzzing	Malformed policies lead to system failures. Fuzz testing is required to check the correct behaviour of the system when an incorrect policy is processed.

REQ-UC1-IKM-3	Support for standard APIs and protocols in infrastructure-key management	Correct protocol implementation		5	Denial of service, leakage of infrastructure information	10	-Fuzzing -Vulnerability assessment -Penetration testing	Mistakes in the security configuration of APIs lead to security mistakes and information leakage. Scanning for common security mistakes and testing the system computers, network devices and applications to see whether they are vulnerable to common attacks. Fuzz testing is required to check the correct behaviour of the systems processing the information received through the APIs while testing is required to evaluate the exposure to potential threats affecting the servers providing the APIs.
REQ-UC1-IKM-4	Support for secure deletion of cryptographic material		Secure storage for keys	6	Keys exposed, data readable	10	-Vulnerability assessment -Penetration testing	Secured storage keys is one of the main requirements for any system. Testing the system to see whether the keys can be exposed due to the deletion of cryptographic material is required.
REQ-UC1-TKM-1	CRUD operations for tenant keys	Trusted tenant administrator	Key existence, trusted infrastructure administrator	7	Confidentiality violation for tenant data	10	-Vulnerability assessment -Penetration testing	Databases protecting tenant data are subject to threats and vulnerabilities (depending on the technology used). Tests related to the potential vulnerabilities are required to ensure the confidentiality of the data and the validity of the keys used.
REQ-UC1-TKM-1	CRUD operations for tenant keys	Secure storage for keys		3	Confidentiality violation for tenant data	10	-Vulnerability assessment -Penetration testing	Databases protecting tenant data are subject to threats and vulnerabilities (depending on the technology used). Tests related to the potential vulnerabilities are required to ensure the confidentiality of the data and the validity of the keys used.
REQ-UC1-TKM-1	CRUD operations for tenant keys	Key length sufficient		3	Confidentiality violation for tenant data	10	-Vulnerability assessment -Penetration testing	Databases protecting tenant data are subject to threats and vulnerabilities (depending on the technology used). Tests related to the potential vulnerabilities are required to ensure the confidentiality of the data and the validity of the keys used.

REQ-UC1-TKM-2	Policy-driven and automated tenant-key management	Policy language complete	Policy engine functional, automation present	4	Policies leak information about tenant keys, expose tenant keys	8	-Fuzzing	Malformed policies lead to system failures. Fuzz testing is required to check the correct behaviour of the system when an incorrect policy is processed.
REQ-UC1-TKM-3	Support for standard APIs and protocols in tenant-key management	Correct protocol implementation		5	Denial of service, leakage of tenant information	10	-Fuzzing -Vulnerability assessment -Penetration testing	Mistakes in the security configuration of APIs lead to security mistakes and information leakage. Scanning for common security mistakes and testing the system computers, network devices and applications to see whether they are vulnerable to common attacks. Fuzz testing is required to check the correct behaviour of the systems processing the information received through the APIs while testing is required to evaluate the exposure to potential threats affecting the servers providing the APIs.
REQ-UC1-TKM-4	Support for secure deletion of cryptographic material		Secure storage for keys	6	Keys exposed, data readable	10	-Vulnerability assessment -Penetration testing	Secured storage keys is one of the main requirements for any system. Testing the system to see whether the keys can be exposed due to the deletion of cryptographic material is required.
REQ-UC1-SKM-1	Redundancy and fault-tolerance in key-management systems	Redundancy and fault-tolerance		2	Denial of service	6	-Vulnerability assessment -Fuzzing	Scanning for common security mistakes and testing the system computers, network devices and applications to see whether they are vulnerable to common attacks.
REQ-UC1-SKM-2	Scalable design of key-management system	System supports actual number of requests		2	Denial of service	6	-Vulnerability assessment -Fuzzing	Scanning for common security mistakes and testing the system computers, network devices and applications to see whether they are vulnerable to common attacks.
REQ-UC1-SKM-3	Key-management solutions support weakly consistent operations in cloud platform	System correctly uses active keys		4	Older keys exposed	4	-Vulnerability assessment -Penetration testing	Testing the system to see whether the system correctly uses active keys.

REQ-UC1-SKM-3	Key-management solutions support weakly consistent operations in cloud platform	System correctly uses active keys		4	Older keys used, data inaccessible	7	-Vulnerability assessment -Penetration testing	Testing the system to see whether the system correctly uses active keys.

Table 4.1: Use Case 1 requirements and their direct and indirect assumptions as well as the suggested security testing for each requirement.

Requirement Reference	Requirement Description	Direct Assumptions	Indirect Assumptions	Violation Likelihood (1/Low-10/High)	Assumed Impact	Threat Severity (1/Low-10/High)	Advocated Security Testing Technique	Comments on Applicability of Security Testing
REQ-UC2-AC1	Access Control per Client	User Authentication		5	Basic assumption violated	8	-Vulnerability assessment -Penetration testing	Only authorized clients have access to the system. Scanning for common security vulnerabilities is required. Furthermore, testing the system computers, network devices and applications to see whether unauthorized users can gain access to the system.
REQ-UC2-AC2	Access control per group of clients	User Groups exist		3	User inconvenience	1	-Vulnerability assessment -Penetration testing	Only authorized clients have access to the system. Scanning for common security vulnerabilities is required. Furthermore, testing the system computers, network devices and applications to see whether unauthorized users can gain access to the system.
REQ-UC2-AC3	Access control per database cell	Cells are controllable		3	Exposure of larger structures, e.g. tables or columns, potentially resulting in partial confidentiality violation	4	-Vulnerability assessment -Penetration testing	Only authorized clients have access to the database cells. Scanning for common security vulnerabilities is required. Furthermore, testing the system to see whether unauthorized users can gain access to the system.
REQ-UC2-AC4	Access control matrix model	Independent of time and workflow		4	User inconvenience	1	-Vulnerability assessment -Penetration testing	Only authorized clients have access to the system. Scanning for common security vulnerabilities is required. Furthermore, testing the system to see whether unauthorized users can gain access to the system.

REQ-UC2-AC5	Access grant and revoke by administrator	Administrator for group maintenance available		3	User inconvenience	1	-Vulnerability assessment -Penetration testing	Only authorized clients have access to the system and only administrator can add/revoke clients. Testing the system to see whether: (i) unauthorized users can gain access to the system and/or (ii) authorized users can act as administrator.
REQ-UC2-AC6	Access control enforced by client	Trusted Client		5	Confidentiality violation for user data	5	-Vulnerability assessment -Penetration testing	Only authorized clients have access to the system and only administrator can add/revoke clients. Testing the system to see whether: (i) unauthorized users can gain access to the system and/or (ii) authorized users can act as administrator.
REQ-UC2-KM1	One key per client	Key length sufficient, keys are renewable	Secure Storage for Key	2	Keys guessable, global confidentiality violation	10	-Penetration testing	One unique key per client is required and the keys should be renewable (fresh keys) and securely stored. Testing the system to check whether the keys can be exposed and if expired keys can be used to gain access the system.
REQ-UC2-KM2	Group key management	Group can secure key, Key is renewable, key length is sufficient		8	Confidentiality violation for group data	5	-Penetration testing	Testing the system to check whether the keys can be exposed or guessable and if the expired keys can be used to gain access the system.
REQ-UC2-KM3	Client key securely stored at client only	Secure Storage		2	Confidentiality violation for user data	5	-Penetration testing	Testing the system to check if Client key stored at the client can be accessible by unauthorized entity.
REQ-UC2-KM4	Group keys derivable	Cryptographic assumptions hold		3	Disaster, global confidentiality violation	10	-Penetration testing	Testing the system to check whether the keys are guessable or exposed.

REQ-UC2-EQ1	Encryption schemes	Cryptographic assumptions hold	Key is secret (i.e. KM1)	3	Disaster, global confidentiality violation	10	-Fuzzing -Vulnerability assessment -Penetration testing	Fuzz testing is required to to check the correct behaviour of the system when the encryption schemes and cryptographic assumptions do not hold. Furthermore, scanning and penetration tests are required to ensure the confidentiality of the data and the validity of the encryption schemes and keys used.
REQ-UC2-EQ2	Adjustable onion encryption	Set of supported queries is sufficient, Scalability is given		4	Performance problems	2	-Vulnerability assessment -Penetration testing	Scanning and penetration tests are required to ensure the confidentiality of the data and the validity of the onion encryption scheme used.
REQ-UC2-EQ3	Proxy re-encryption, Query rewriting, Post-processing	Security enforcement is possible for Multi user environment		3	User has access to data which was supposed to be revoked, resulting in partial confidentiality violation	4	-Fuzzing	Fuzz testing is required to to check the correct behaviour of the system when an incorrect policy is processed.
REQ-UC2-EQ4	Support for different keys	Set of supported queries is sufficient for Multi user, Scalability is given for Multi user		5	Performance problems	2	-Fuzzing	Fuzz testing is required to to check the correct behaviour of the system when different keys are supported for Multi user.

Table 4.2: Use Case 2 requirements and their direct and indirect assumptions as well as the suggested security testing for each requirement.

Requirement Reference	Requirement Description	Direct Assumptions	Indirect Assumptions	Violation Likelihood (1/Low-10/High)	Assumed Impact	Threat Severity (1/Low-10/High)	Advocated Security Testing Technique	Comments on Applicability of Security Testing
REQ-UC3-KM-1	Each tenant should be provisioned with an instance of a key management service from the cloud service store	Tenant has an account on the cloud service store	Tenant has the ability to subscribe to the DPaaS	1	Denial of Service	2	-Vulnerability assessment -Fuzzing	Scanning for vulnerabilities as well as Fuzz testing is required to check the correct behaviour of the system (checking whether each tenant is provisioned with an instance of a key management). Furthermore, to verify whether the system is vulnerable to common attacks, such as enumeration of security related information and denial of service attacks.
REQ-UC3-KM-2	The tenants should be able to generate, insert, retrieve and remove keys from their key management service	Tenant has access to the KMS	KMS is operational	2	Denial of Service	2	-Fuzzing	Fuzz testing is required to check the tenants ability to generate, insert, retrieve and remove keys from their key management service as specified in the system behaviour.
REQ-UC3-KM-3	The key management service should be able to offer different key types and generation algorithms to each tenant, e.g., AES128, AES256, 3DES etc.	KMS is operational and available		1	Denial of Service	2	-Fuzzing -Penetration testing	Checking the type of encryption algorithm as well as the key sizes is essential to protect the system from different types of attacks. This checking is performed using the penetration testing, in order to test whether the system is vulnerable to common attacks using each algorithm. Fuzzing testing is used to check the correct behaviour of the system.

REQ-UC3-KM-4	Only the tenants should be able to create and manage the keys	Multi-tenant feature of the KMS is operational and available		2	Confidentiality and Integrity violation of keys	10	-Fuzzing -Vulnerability assessment -Penetration testing	Fuzz testing is required to check the tenants ability to create and manage keys from their key management service as specified in the system behaviour. Furthermore, scanning and penetration testing are used to check if only the tenants can create and manage keys or not.
REQ-UC3-KM-5	The cloud service providers should have no access or visibility of the tenants' keys	KMS is hosted in a trusted environment	KMS is in a hardened OS or Sandbox	1	Confidentiality and Integrity violation of keys	10	-Vulnerability assessment -Penetration testing	Scanning and penetration testing are used to check the provider's ability to access the tenants' keys which leads to Confidentiality and Integrity violation of the keys and the tenants' data as well.
REQ-UC3-KM-6	The tenants should be able to cache their keys on trusted virtual machines or gateways in order to outsource or improve performance of the encryption and decryption process	KMS can generate cache-able keys, unique to particular hosts		2	Degradation of encryption and decryption performance	1	-Fuzzing	Tests are required to check the tenants ability to cache their keys on trusted virtual machines or gateways in order to outsource or improve performance of the encryption and decryption process as specified in the system behaviour.
REQ-UC3-AC-1	Each tenant should be provisioned with an instance of an access control service from the cloud service store	Tenant has an account on the cloud service store	Tenant has the ability to subscribe to the DPaaS	2	Denial of Service	2	-Fuzzing	Tests are used to check that each tenant is provisioned with an instance of an access control service.

REQ-UC3-AC-2	The tenants should be able to create, delete and modify access control policies from their instance of the access control service	Tenant has access to the AC service	AC service is operational	2	Denial of Service	2	-Fuzzing	Tests are required to check the tenants ability to cache their keys on trusted virtual machines or gateways in order to outsource or improve performance of the encryption and decryption process as specified in the system behaviour.
REQ-UC3-AC-3	The access control service should be able to offer use of different system and data attributes for the construction of a security rule, e.g., file-system, user, application, and time attributes.	AC service is operational and available		1	Denial of Service	2	-Fuzzing	Fuzz testing is required to check the correct behaviour of the system when an incorrect access control service is processed.
REQ-UC3-AC-4	Only the tenants should be able to create and manage their access control policies	Multi-tenant feature of the AC service is operational and available		2	Confidentiality and Integrity violation of keys	10	-Fuzzing -Vulnerability assessment -Penetration testing	Fuzz testing is required to check the tenants ability to create and manage keys from their their access control policies and also to check the correct behaviour of the system when an incorrect policy is processed. Furthermore, scanning and penetration testing are used to check if only the tenants can create and manage their access control policies or not.

REQ-UC3-AC-5	The cloud service providers should have no access or visibility of the tenants' access control policies	AC service is hosted in a trusted environment	AC service is in a hardened OS or Sandbox	2	Confidentiality and Integrity violation of keys	10	-Vulnerability assessment -Penetration testing	Scanning and penetration testing are used to check the provider's ability to access the tenants' access control policies which leads to confidentiality and Integrity violation of the keys and the tenants' data as well.
REQ-UC3-AC-6	All data protection operations should be governed by access control policies by either approving or denying access to the required keys	Key release is tightly coupled with AC service		5	Partial loss of data access - Denial of Service	8	-Fuzzing	Fuzz testing is required to check the correct behaviour of the system when an incorrect policy is processed.
REQ-UC3-AC-7	The access control service of tenants should be tightly coupled with their key management service, such that no key can be utilised without an approving access control policy	Key release is not possible without correct policy		5	Partial loss of data access - Denial of Service	8	-Fuzzing	Fuzz testing is required to check the correct behaviour of the system when an incorrect access control policy is processed.
REQ-UC3-SO-1	Each tenant should be provisioned with a cloud service store account	Each tenant is given a unique identifier		2	Denial of Service	2	-Fuzzing	Tests are required to check that each tenant is provisioned with a cloud service store.

REQ-UC3-SO-2	The service store should provide the tenants with access to the storage services of multiple cloud service providers	Service store has access profiles of relevant cloud service provider		3	Denial of Service, Partial loss of data access	3	-Fuzzing	Tests are required to check that tenants ability to access the storage services of multiple cloud service providers as specified in the system behaviour.
REQ-UC3-SO-3	The service store should be able to offer block storage service to the tenants	Service store is able to provision block storage from relevant cloud service provider		3	Denial of Service, Partial loss of data access	3	-Fuzzing	Tests are required to validate that the service store is able to provision block storage from relevant cloud service provider.
REQ-UC3-SO-4	The service store should be able to offer object storage service to the tenants	Service store is able to provision object storage buckets from relevant cloud service provider		3	Denial of Service, Partial loss of data access	3	-Fuzzing	Tests are required to validate that the service store is able to provision object storage buckets from relevant cloud service provider.
REQ-UC3-SO-5	The service store should be able to offer Big Data storage service (HDFS) to the tenants	Service store is able to provision HDFS service from relevant cloud service provider		3	Denial of Service, Partial loss of data access	3	-Fuzzing	Tests are required to validate that the service store is able to provision HDFS service from relevant cloud service provider.
REQ-UC3-SO-6	The tenants should be able to enable or disable the use of data protection service on the storage service of their choice	Service store can subscribe or unsubscribe from the DPaaS		1	Denial of Service	5	-Fuzzing	Tests are required to check the tenants' ability to enable or disable the use of data protection service on the storage service of their choice according to the system behaviour.

REQ-UC3-SO-7	The service store should be able to offer key management as a service to the tenants	KMS has an operational plug-in for the Service Store		4	Denial of Service	10	-Fuzzing	Tests are required to validate that the service store is able to offer key management as a service to the tenants.
REQ-UC3-SO-8	The service store should be able to offer access control as a service to the tenants	AC Service has an operational plug-in for the Service Store		4	Denial of Service	10	-Fuzzing	Tests are required to validate that the service store is able to offer access control as a service to the tenants as specified in the system behaviour.
REQ-UC3-DE-1	The core encryption process should only be controlled and managed by the tenant	Only the tenant can specify the target storage services where data is stored		4	Confidentiality and Integrity violation of tenant data	10	-Vulnerability scanning -Penetration testing	Scanning and penetration testing are used to check if only the tenants can control and manage the keys to eliminate confidentiality and integrity violation of the tenant data.
REQ-UC3-DE-2	The tenant should be able to deploy and manage the core encryption process on trusted virtual machines or gateways as an agent or plug-in	Service store has configuration management capability for VMs and gateways	An agent or plug-in for target VMs or gateways	4	Denial of Service	5	-Fuzzing -Penetration testing	Tests are required to validate that the tenant is able to deploy and manage the core encryption process on trusted virtual machines or gateways as an agent or plug-in.
REQ-UC3-DE-3	The core encryption process should be FIPS 140 compliant	DPaaS components are compliant to industry standards		3	Possible confidentiality and integrity violation due to weak algorithms	7	-Fuzzing	Fuzz testing is required to check the correct behaviour of the system when an incorrect policy is processed.

REQ-UC3-DE-4	The encryption agent or plug-in should be able to access the tenant's key management service and access control service	These services are accessible over an encrypted channel using protocols like SSL/TLS		4	Loss of confidentiality, integrity and availability	10	-Vulnerability assessment -Penetration testing	Tests are required to check the accessibility of the services from unauthorized agent or tenants.
REQ-UC3-DE-5	The keys should only be released to the encryption agent or plug-in upon approval of an access control policy	Key release is not possible without correct policy		5	Partial loss of data access - Denial of Service	8	-Fuzzing	Fuzz testing is required to check the correct behaviour of the system when an incorrect policy is processed.

Table 4.3: Use Case 3 requirements and their direct and indirect assumptions as well as the suggested security testing for each requirement.

Requirement Reference	Requirement Description	Direct Assumptions	Indirect Assumptions	Violation Likelihood (1/Low-10/High)	Assumed Impact	Threat Severity (1/Low-10/High)	Advocated Security Testing Technique	Comments on Applicability of Security Testing
REQ-UC4-AC1	Access Control to the web portal	Proper user authentication and proper rights for the logged user	Corporate password policy enforcement	5	Data confidentiality is violated	8	-Fuzzing -Vulnerability assessment -Penetration testing	Authenticated users only should be allowed to log to the system. Tests related to the potential vulnerabilities are required to ensure the proper user authentication and proper rights for the logged user.
REQ-UC4-AC2	Save credentials in the device	Credentials accessible only by the owner		3	Data confidentiality is violated	8	-Fuzzing -Vulnerability assessment -Penetration testing	Testing the system to see whether the data is securely stored and encrypted by the owner.
REQ-UC4-AC3	Access control to middleware	Proper user authentication and proper rights for the logged user	Corporate password policy enforcement	5	Data confidentiality is violated	8	-Fuzzing -Vulnerability assessment -Penetration testing	Authenticated users only should be allowed to log to the system. Tests related to the potential vulnerabilities are required to ensure the proper user authentication and proper rights for the logged user.
REQ-UC4-AC4	Access to shared files only with permission of the file owner	Only exist a file owner		3	Shared file confidentiality is violated	4	-Fuzzing -Vulnerability assessment -Penetration testing	Tests related to the potential vulnerabilities are required to ensure that access to shared files only with permission of the file owner and thus ensuring the confidentiality of the data.
REQ-UC4-AC5	Access grant by administrator for locked users	Administrator is the only entity that can assign permission	Malicious user was previously blocked	3	User has access to data which was supposed to be revoked, resulting in partial confidentiality violation	7	-Fuzzing -Penetration testing	Scanning and penetration testing are used to check whether non-permitted users have the ability to access the data, which leads to confidentiality and integrity violation of the data .

REQ-UC4-AC6	Limit of failed attempts	There is a maximum number of attempts	Key is sufficiently safe	2	User data confidentiality is violated	8	-Fuzzing -Vulnerability assessment -Penetration testing	Testing the system to see whether the system correctly uses active keys.
REQ-UC4-SS1	Ensure Cloud capacity	Check available capacity in the Cloud before store user data		3	Failure to upload user data	6	-Fuzzing	Testing the system to see whether the Cloud capacity can be adapted to meet user requirements.
REQ-UC4-SS2	Storage access control through middleware	Users cannot access Cloud storage bypassing the middleware		3	Total access to storage systems	10	-Fuzzing	
REQ-UC4-SS3	Eastic Cloud capacity	Capacity can be adapted to meet user requirements	There is sufficient capacity in the Cloud	2	User data cannot be stored in the Cloud	6	-Fuzzing	Testing the system to see whether the Cloud capacity can be adapted to meet user requirements.
REQ-UC4-SS4	Comply data protection directive (EU 95/46/EC)	There are enough mechanisms to protect user personal data		2	Disclosure of personal data	7	-Fuzzing	Fuzz testing is required to check the correct behaviour of the system when an incorrect policy is processed.
REQ-UC4-SS5	Data recovery control	Exist backup mechanism in the system	The backup system is resilient	2	Loss of user data	6	-Vulnerability assessment -Fuzzing	
REQ-UC4-DE1	Only data owner can decrypt data	Data are encrypted in the Cloud and user possesses the encryption key		3	User data confidentiality violation	9	-Vulnerability assessment -Penetration testing	Testing the system to see whether the system correctly uses active keys and only the owner can decrypt data.
REQ-UC4-DE2	Store data encrypted in the Cloud	Client side encryption		1	User data confidentiality violation	6	-Vulnerability assessment -Penetration testing	Testing the system to see whether the system correctly uses active keys and data is securely stored and encrypted in the Cloud.

REQ-UC4-DE3	Server synchronization before decryption	Prior stage to user-data interaction		1	User cannot access its data	3	-Fuzzing -Vulnerability assessment -Penetration testing	Testing is performed to check whether files are up to date and accessed by the user.
REQ-UC4-DE4	Secured file download through the web browser	Client side decryption		1	Data confidentiality, integrity and availability violation	8	-Fuzzing -Vulnerability assessment -Penetration testing	Testing is performed to check whether files can be accessed during file downloading, which leads to confidentiality and integrity violation of the data.

Table 4.4: Use Case 4 requirements and their direct and indirect assumptions as well as the suggested security testing for each requirement.

5. Conclusion

There is an increasing emphasis on security testing, along with a multitude of testing techniques being proposed in the recent years. This growth is supported by the growing complexity of cloud systems, by the corresponding increasing number of threats, by the increasing dependence on cloud based services and by the increasing user-sensitive information that cloud services store and process. However, several obstacles impede the adoption of security testing techniques in cloud environments. In the pre-cloud times, services (and the data managed by them) were normally under the control of their users. For example, it was common that companies managed their own databases. More recently, companies are outsourcing the management of some of their traditional services. While this reduces costs and increases performance it also entails the loss of control as it is the cloud providers (and not customers) who are responsible for managing services and data. This is even more important for multi-cloud services where different clouds, often from different companies, collaborate in the cloud service provisioning.

In such new scenarios of cloud and multi-cloud services, security testing techniques become essential to guarantee the security of customers services and data, in order to protect them from potential threats. However, using these techniques becomes a challenge, precisely for the lack of control on the services and data. The choice of what technique to use highly depends on the characteristics of the cloud services (i.e., the resources available, the type of service provided). To this end cloud service providers are often quite reluctant to open their systems to perform tests. White-box testing techniques are only suitable when the source code is available. Black box tests are truly viable in the cloud context as long as interfaces are normally available. However, in many cases not even the interfaces are available and only application level testing can be performed.

In this deliverable we have addressed the problem of the applicability of security testing techniques according to the availability of cloud resources. We have used this analysis to evaluate the use cases elicited in ESCUDO-CLOUD and the requirements associated to them. Our security testing methodology considers all UC requirements to extract a range of aspects therein - the related resources, the associated security aspects (e.g., confidentiality or availability), the impact in case of unfulfillment of requirements, etc. Subsequently, the range of available security testing techniques are evaluated and mapped to these requirement to advocate the most appropriate testing technique to use in each use case.

Bibliography

[AAD+09] Jeff Arnold, Tim Abbott, Waseem Daher, Gregory Price, Nelson Elhage, Geoffrey Thomas, and Anders Kaseorg. Security impact ratings considered harmful. *In Proc. of Hot Topics in Operating Systems*, 2009.

[ABB+05] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, et al. The avispa tool for the automated validation of internet security protocols and applications. In *International Conference on Computer Aided Verification*, pages 281–285, 2005.

[AFR02] Jean Arlat, Jean-Charles Fabre, and Manuel Rodríguez. Dependability of COTS Microkernel-Based Systems. 51(2):138–163, 2002.

[Ait02] Dave Aitel. An introduction to spike. In *Proc. of BlackHat USA*, 2002.

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Transactions on Dependable and Secure Computing*, volume 1, pages 11–33, 2004.

[BBC+13] Julien Botella, Fabrice Bouquet, Jean-François Capuron, Franck Lebeau, Bruno Legeard, and Florence Schadle. Model-based testing of cryptographic components–lessons learned from experience. In *Proc. of International Conference on Software Testing, Verification and Validation*, pages 192–201, 2013.

[BBGM12] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *Proc. of International Conference on Software Testing, Verification and Validation*, pages 818–825, 2012.

[BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proc. of 11th ACM conference on Computer and Communications Security*, pages 132–145, 2004.

[BCL08] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. *In Proc. of International Conference on Trusted Computing*, pages 166–178, 2008.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proc. of USENIX Annual Technical Conference*, pages 41–46, 2005.

[BNE16] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. In *Proc. of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–157, 2016.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, volume 8, pages 209–224, 2008.

[CERa] CERT. Cert basic fuzzing framework. `https://www.cert.org/vulnerability-analysis/tools/bff.cfm`.

[CERb] CERT. Cert failure observation engine. `https://www.cert.org/vulnerability-analysis/tools/foe.cfm`.

[CGN+13] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor Trivedi. Fault Triggers in Open-Source Software: An Experience Report. In *Proc. of International Symposium on Software Reliability Engineering*, pages 178–187, 2013.

[Che09] Liqun Chen. A daa scheme requiring less tpm resources. *Proc. of International Conference on Information Security and Cryptology*, pages 350–365, 2009.

[CPS10] Liqun Chen, Dan Page, and Nigel Smart. On the design and implementation of an efficient daa scheme. *In Proc. of International Conference on Smart Card Research and Advanced Applications*, pages 223–237, 2010.

[CVE14] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160, 2014.

[CVE16] CVE-2016-2208. Available from MITRE, CVE-ID CVE-2016-2208, 2016.

[CWB15] Sang Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proc. of Symposium on Security and Privacy*, pages 725–741, 2015.

[DCBM06] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. GridUnit: Software Testing on the Grid. In *Proc. of the International Conference on Software Engineering*, pages 779–782, 2006.

[DHK11] Frédéric Dadeau, Pierre-Cyrille Héam, and Rafik Kheddam. Mutation-based test generation from security protocols in hlpsl. In *Proc. of International Conference on Software Testing, Verification and Validation*, pages 240–248, 2011.

[DLAS+12] Domenico Di Leo, Fatemeh Ayatolahi, Behrooz Sangchoolie, Johan Karlsson, and Roger Johansson. On the Impact of Hardware Faults–An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions. In *Proc. of International Conference on Computer Safety, Reliability, and Security*, pages 198–209. 2012.

[DMK10] Huning Dai, Christian Murphy, and Gail Kaiser. Configuration fuzzing for software vulnerability detection. In *Proc. of Availability, Reliability, and Security*, pages 525–530, 2010.

[FCKV10] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proc. of USENIX Security Symposium*, volume 58, 2010.

[FIR15] FIRST.Org, Inc. (FIRST). Common Vulnerability Scoring System v3.0: Specification Document. https://www.first.org/cvss/cvss-v30-specification-v1.7.pdf, 2015.

[FM09] Christian Fruhwirth and Tomi Mannisto. Improving cvss-based vulnerability prioritization and response with context information. In *Proc. of International Symposium on Empirical Software Engineering and Measurement*, pages 535–544, 2009.

[FZB+16] Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler, and Alexander Pretschner. Model-based security testing: a taxonomy and systematic classification. In *Software Testing, Verification and Reliability*, volume 26, pages 119–148, 2016.

[GDJ+11] Haryadi Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph. Hellerstein, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proc. of the Symposium on Networked Systems Design & Implementation*, 2011.

[GKL08] Patrice Godefroid, Adam Kiezun, and Michael Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215, 2008.

[GLM08] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *Proc. of Network & Distributed System Security Symposium*, volume 8, pages 151–166, 2008.

[GLR09] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proc. of International Conference on Software Engineering*, pages 474–484, 2009.

[GM01] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In *CryptographersŤrack at the RSA Conference*, pages 425–440. Springer, 2001.

[Goo] Google. syzkaller. https://github.com/google/syzkaller.

[GPC+03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206, 2003.

[Gro] NCC Group. TriforceAFL. https://github.com/nccgroup/TriforceAFL.

[HB10] Pete Herzog and Marta Barcelo. Osstmm 3 – the open source security testing methodology manual, 2010.

[HHZ12] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Poc. of USENIX Security Symposium*, pages 445–458. USENIX, 2012.

[Hoc] S Hocevar. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf.

[Hof15] Jörg Hoffmann. Simulated penetration testing: From "dijkstra" to "turing test++". In *Proc. of International Conference on Automated Planning and Scheduling*, pages 364–372, 2015.

[HSNB13] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proc. of USENIX Security Symposium*, pages 49–64, 2013.

[JGS11] Pallavi Joshi, Haryadi Gunawi, and Koushik Sen. PREFAIL: A Programmable Tool for Multiple-failure Injection. In *Proc. of ACM SIGPLAN Notices*, volume 46, pages 171–188, 2011.

[JH08] Yue Jia and Mark Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Proc. of Source Code Analysis and Manipulation*, pages 249–258, 2008.

[JNB16] Sadeeq Jan, Duy Nguyen, and Lionel Briand. Automated and effective testing of web services for xml injection attacks. In *Proc. of the International Symposium on Software Testing and Analysis*, 2016.

[KD00] Phil Koopman and John DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. In *IEEE Transactions on Software Engineering*, volume 26, pages 837–848, 2000.

[Laa11] Wladimir Laan. dropship - dropbox api utilities. `https://github.com/driverdan/dropship`, 2011.

[Las05] Alexey Lastovetsky. Parallel testing of distributed software. In *Information and Software Technology*, volume 47, pages 657 – 662, 2005.

[LNW+14] Anna Lanzaro, Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. An Empirical Study of Injected versus Actual Interface Errors. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 397–408, 2014.

[McD00] James McDermott. Attack net penetration testing. In *Proc. of the Workshop on New Security Paradigms*, pages 15–21, 2000.

[MFS90] Barton Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. In *Communications of the ACM*, volume 33, pages 32–44, 1990.

[MM14] Matteo MEUCCI and Andrew Muller. The owasp testing guide 4.0. 2014.

[MNC16] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Dekant: A static analysis tool that learns to detect web application vulnerabilities. In *Proc. of the International Symposium on Software Testing and Analysis*, 2016.

[MP07] Charlie Miller and Zachary Peterson. Analysis of mutation and generation-based fuzzing. *White Paper, Independent Security Evaluators, Baltimore, Maryland*, 2007.

[MR05] Christoph Michael and Will Radosevich. Risk-based and functional security testing. *In Build Security*, 2005.

[MSL+11] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proc. of USENIX Security Symposium*, pages 65–76, 2011.

[Nat08] National Institute Of Standards and Technology. *NIST Special Publication 800-115*. 2008.

[Nat11] National Institute Of Standards and Technology. *NIST Special Publication 800-53A*. 2011.

[NCDM13] Roberto Natella, Domenico Cotroneo, Joao Duraes, and Henrique Madeira. On Fault Representativeness of Software Fault Injection. In *IEEE Transactions on Software Engineering*, volume 39, pages 80–96, 2013.

[NIS08] NIST. Sp 800-66 revision 1: An introductory resource guide for implementing the health insurance portability and accountability act (hipaa) security rule, 2008.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, pages 44–66, 2003.

[Oeh05] Peter Oehlert. Violating assumptions with fuzzing. In *IEEE Security & Privacy*, volume 3, pages 58–62, 2005.

[OU10] Manuel Oriol and Faheem Ullah. YETI on the Cloud. In *Proc. of Software Testing, Verification, and Validation Workshops*, pages 434–437, 2010.

[OWA16] OWASP. Application security verification standard 3.0.1. `https://www.owasp.org/images/3/33/OWASP_Application_Security_Verification_Standard_3.0.1.pdf`, 2016.

[PCI16a] PCI. Payment card industry data security standard (pci-dss): Requirements and security assessment procedures v3.2. `https://www.pcisecuritystandards.org`, 2016.

[PCI16b] PCI Security Standards Council. Pci data security standard v3.2, 2016.

[Pen15] Penetration Test Guidance Special Interest Group, PCI Security Standards Council. Penetration testing guidance, 2015.

[PPB03] Thomas Peltier, Justin Peltier, and John Blackley. Information security fundamentals. 2003.

[PSD+06] Ronald Perez, Reiner Sailer, Leendert Doorn, Stefan Berger, Ramón Cáceres, and Kenneth Goldman. vtpm: virtualizing the trusted platform module. In *Proc. of USENIX Security Symposium*, pages 305–320, 2006.

[RCA+14] Alexandre Rebert, Sang Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proc. of USENIX Security Symposium*, pages 861–875, 2014.

[RM11] Sanjay Rawat and Laurent Mounier. Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results. In *Proc. of Software Testing, Verification and Validation Workshops*, pages 531–533, 2011.

[Rud07] Jesse Ruderman. Introducing jsfunfuzz. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz`, 2007.

[SAM08] Akbar Siami, James Andrews, and Duncan Murdoch. Sufficient Mutation Operators for Measuring Test Effectiveness. In *Proc. of International Conference on Software Engineering*, pages 351–360, 2008.

[Sch99] Bruce Schneier. Attack trees. In *Dr. Dobb's journal*, volume 24, pages 21–29, 1999.

[SDLR+15] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *ACM SIGPLAN Notices*, volume 50, pages 473–486, 2015.

[SGR09] Nuno Santos, Krishna Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. *In Proc. of HotCloud*, 9:3–3, 2009.

[SGS12] Ina Schieferdecker, Juergen Grossmann, and Martin Schneider. Model-based security testing. In *Proc. of Model-Based Testing*, pages 1–12, 2012.

[SGS+16] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proc. of the Network and Distributed System Security Symposium*, 2016.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dong Song. Sok: Eternal war in memory. In *Proc. of Security and Privacy*, pages 48–62, 2013.

[SSSW98] Chris Salter, Sami Saydjari, Bruce Schneier, and Jim Wallner. Toward a secure system engineering methodolgy. In *Proc. of the Workshop on New security Paradigms*, pages 2–10, 1998.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. *In IEEE Symposium on Security and Privacy*, pages 138–157, 2016.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. In *Software Testing, Verification and Reliability*, volume 22, pages 297–312, 2012.

[WAW09] Martin Weiglhofer, Bernhard Aichernig, and Franz Wotawa. Fault-based conformance testing in practice. In *Interntional Journal Software and Informatics*, volume 3, pages 375–411, 2009.

[WCGB13] Maverick Woo, Sang Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proc. of ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[Wei73] Clark Weissman. *System security analysis/certification methodology and results*. System Development Corporation, 1973.

[WPS+15] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. Grinder: On reusability of fault injection tools. In *Proc. of the International Workshop on Automation of Software Test*, pages 75–79, 2015.

[WSN+15] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. No pain, no gain?: The utility of parallel fault injections. In *Proc. of the International Conference on Software Engineering*, pages 494–505, 2015.

[WTSS13] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. simFI: From single to simultaneous software fault injections. In *Proc. of International Conference on Dependable Systems and Networks*, pages 1–12, 2013.

[WWGZ11] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. In *ACM Transactions on Information and System Security*, volume 14, pages 1–28, 2011.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294, 2011.

[Zal] Michael Zalewski. American Fuzzy Lop (AFL). `http://lcamtuf.coredump.cx/afl`.

[Zec11] Philipp Zech. Risk-based security testing in cloud computing environments. In *Proc. of International Conference on Software Testing, Verification and Validation*, pages 411–414, 2011.