Security Testing beyond Functional Tests

Mohammad Torabi Dashti and David Basin

Department of Computer Science
ETH Zurich

Abstract. We present a theory of security testing based on the basic distinction between system specifications and security requirements. Specifications describe a system's desired behavior over its interface. Security requirements, in contrast, specify desired properties of the world the system lives in. We propose the notion of a security rationale, which supports reductive security arguments for deriving a system specification and assumptions on the system's environment sufficient for fulfilling stated security requirements. These reductions give rise to two types of tests: those that test the system with respect to its specification and those that test the validity of the assumptions about the adversarial environment. It is the second type of tests that distinguishes security testing from functional testing and defies systematization and automation.

1 Introduction

Security testing plays an essential role in quality assurance for information technologies ranging from traditional software applications to cyber-physical control systems. Various security testing tools and techniques are available today, and a wide range of systems are regularly subjected to security tests. Yet, the literature lacks the necessary frame of reference to articulate and answer basic questions regarding security testing. For example, most practitioners would agree that security testing is harder than functional testing, measuring the adequacy of security tests is challenging, and some kinds of security testing, such as penetration testing, defy systematization and automation. However, there exists no coherent explanation for these phenomena.

We rationally reconstruct security testing around the notion of security requirements. Our starting point is the key distinction between system specifications and security requirements. A system specification describes how an artifact, or system, must behave in an environment. A security requirement, in contrast, expresses desired properties of the environment controlled by the system. Consider, for example, an office. A system specification for an electronic lock installed as part of the office's door might state that the lock opens the door if and only if a valid key is presented. A security requirement might be that access to the office is restricted to employees working there. Under certain assumptions, if the system (here, the lock) satisfies its specification, then the requirement is satisfied in the actual environment. In our example, these environmental assumptions include: the office has no entrance other than the door controlled by the lock, and only those working in the office have a valid key. What distinguishes security requirements from other requirements is that they must hold in the presence of an adversary. In the office example, if the adversary can climb through an open window,

then the requirement is violated, regardless of whether or not the deployed lock satisfies its specification. Accounting for the adversary's capabilities is therefore integral to testing security requirements.

We introduce the notion of a security rationale, which supports reductive security arguments for deriving a system specification and the assumptions on the system's environment sufficient for fulfilling stated security requirements. These reductions give rise to two types of tests: (1) those that test the system with respect to its specification and (2) those that test the validity of the assumptions about the adversarial environment. These types sharply distinguish security testing from functional testing. The purpose of functional tests is to refute the hypothesis that the system satisfies its (functional, security, or other) specification; this corresponds to just the first type. In contrast, security testing requires both types of tests. This distinction allows us to precisely explain in what sense security testing is harder than functional testing. It also provides a frame of a reference for delimiting the scope and reach of existing security testing techniques and procedures. We illustrate this point through examples from fuzz testing, fault injection, risk-based security testing, and vulnerability-driven security testing.

We describe why measuring the adequacy of security tests is challenging by demonstrating that security tests are inherently incomplete. This incompleteness, we argue, stems from the open-ended nature of the assumptions that are part of security rationales. It is therefore orthogonal to the incompleteness of functional tests, which is rooted in the infinite cardinality of the domains where test inputs are selected. The open-ended nature of the assumptions also explains why security testing intrinsically depends on the testers' creativity and resources, thereby defying automation and systematization. Finally, we clarify testing and vulnerability remediation procedures associated with our two test types.

The theory of security testing that we develop is novel. The most closely related work comes from the domain of requirements engineering, where one studies the relationship between systems and the domains in which they operate; see, for example, Jackson's world-machine model [13], which inspired our notion of security rationales. Security testing itself is a broad topic and an extended literature review is outside this paper's scope. Nevertheless, we discuss along the way various prominent security testing techniques and procedures, such as [11, 19, 24].

Structure of Paper. We define specifications and requirements in §2. We introduce the notion of a security rationale in §3, which relates specifications and security requirements. Security cases, introduced in §4, justify the conformance of a concrete system to a security rationale in the presence of a specific adversary. In §5, we define two types of security tests whose purpose is to refute the hypothesis that a security requirement is satisfied in an adversarial environment. We also discuss the role of these two types in practice and comment on vulnerability remediation. We draw conclusions in §6.

2 Specifications and Requirements

Our starting point is valuable **resources** worth protecting, such as data on a company's web server or documents in an office. We consider security **requirements** that pertain

to these resources. For instance, if our resources are the books in a library, the security requirement might be that only those possessing a valid library card may borrow books. In general, such requirements reflect the constraints that stakeholders impose on access to the resources.

What distinguishes security requirements from other requirements, such as functional requirements, is that they must hold in the presence of an adversary. The **adversary** (or **threat agent**) is the entity against whom the resources must be protected. Examples of adversaries include disgruntled employees, curious trespassers, and nation-state attackers. A security requirement that is satisfied in the presence of one adversary might not hold in the presence of a more capable one. A resource's security is therefore not a meaningful property without fixing the adversary's capabilities, which is left implicit in the above library example. Risk analysis can for example be used to identify the adversary in whose presence a security requirement must be satisfied.

To satisfy a security requirement, we construct systems and deploy them in the (resource's) environment. A **system** is an artifact whose behaviors can be regulated and controlled. A system affects its environment by interacting with it through an **interface**. To control access to an office, we may for example install an electronic lock system. This system changes the environment by restricting who may enter through the office's door. A **specification** describes the desired behaviors of a system over its interface. For instance, for the lock system, described in more detail in Example 1 below, its specification relates input received by its sensors, e.g., a smart-card reader, with output to its actuators, which control the lock's cylinder. Note that specifications need not directly constrain a system's internal structure. Our electronic lock specification does not for instance express a preference for a particular memory layout for the lock's software.

There is a fundamental distinction between specifications and requirements. Specifications constrain a system's behaviors over its interface. Requirements, in contrast, constrain an adversary's access to resources in an environment. Therefore, requirements neither directly prohibit nor oblige any system behaviors, and systems do not directly guarantee a requirements' satisfaction. The following example illustrates this point.

Example 1. An R&D laboratory contains sensitive documents. To limit access to the documents, an electronic lock system is installed at the lab's door. A security requirement for the documents states that only those staff members working in the lab may read them. This does not prohibit (or oblige) any input-output behavior for the lock. In contrast, a specification for the lock states: the output signal `open` is produced only after receiving as input a `key` that belongs to the set `validKeys`. Here `open` is the signal that, say, triggers an actuator that opens the lock. The satisfaction of this specification, which describes the lock's desired input-output behavior over its interface, does not entail the requirement's satisfaction. The lab may have an open window.

As a side remark, the above requirement is rather weak. It does not, for example, prohibit information flow arising from a careless staff member leaking a document's content to an outsider. This requirement's satisfaction, therefore, does not entail the documents' confidentiality. △

The fundamental distinction between specifications and requirements is at the heart of our development, and we explore its implications in detail. Two comments are however due here. First, while a specification applies to a system's (input-output) interface, a

requirement cannot be attributed with an interface because resources and environments do not have definite interfaces. The following example illustrates this point.

Example 2. A publishing company's database stores data that is subject to the following integrity *requirement*: only copy editors may delete data. Dynamite that explodes in the database's vicinity constitutes an "input" that can delete the data, thereby affecting its integrity. Similarly, formatting the database's storage media or invoking the database's rollback operation are both "inputs" that can also delete the stored data. Clearly the integrity requirement above cannot be attributed to a specific interface.

Now suppose that the database's input-output interface is realized through an API. A *specification* for the database system, which applies to this interface, states: only the users who have the role `copy_editor` may execute the API's `delete` command. An input is a user's identity (or roles) together with the API command the user requests to execute. The system then either executes the command or denies the request. △

Second, a system's interface may consist of multiple communication channels between the system and its environment. A **nominal** channel is a channel that has been anticipated in the system's specification. A trusted computing device, for example, has a nominal channel, realized through its API. A **side channel** is an unanticipated communication channel between the system and its environment, and by extension, the adversary. Measuring a trusted device's power consumption may for instance reveal a secret key stored on the device. Similarly, magnetic fields can degauss, and hence write to, the device's magnetic storage, and row-hammer attacks [16] can write to protected memory locations. These constitute side channels when the device's specification does not describe the device's behavior on these channels. Whether an adversary can exploit a nominal channel or a side channel to communicate with a system depends on the adversary's capabilities and the system's environment.

The relationship between specifications and requirements is central to security design and analysis. The problem security engineers solve is that of satisfying a security requirement in an adversarial environment. A solution to the problem consists of one or more systems that are deployed in the environment; cf. [13]. No single solution however solves all problems, because solutions are invariably contingent upon assumptions about the environment they address. We call these **environmental assumptions**. The following example illustrates this notion.

Example 3. Consider the scenario of Example 1. The lock system addresses the stated security requirement for the documents, provided various environmental assumptions are satisfied. These include that the only way to enter the lab is through the door controlled by the lock system. The adversary's capabilities affect this assumption's validity. Suppose that the lab has a window. If the adversary can climb in through the window, then this environmental assumption, and consequently the requirement, are violated. △

The distinction between specifications and requirements is similar to the distinction between mechanisms and the policies they are intended to enforce [17]. A system is a mechanism that maps symbols to symbols, independently from the environment where it is deployed. This is of course desirable: a XACML policy enforcement point, an AES

encryption module, and a lock system should do what their specifications promise, regardless of where they are deployed. In this sense, systems are just symbol manipulators, oblivious to their deployment environment. In contrast, a resource's security requirements impose certain (access) relations among actual entities in an environment. A symbol-manipulation entity can contribute to security only based on the environmental assumption that its input-output behavior is given an appropriate interpretation; cf. [22]. The following example illustrates this point.

Example 4. Consider the database scenario of Example 2. The database's specification (for its nominal channel) contributes to the security of the data under environmental assumptions, including: only copy editors have the role `copy_editor`, and there is no way to delete the data except by executing the API's `delete` command. These assumptions reflect the condition that the symbolic notions of a role and a command are interpreted appropriately in the context of the given scenario. △

To capture the relationship between requirements, specifications, and environmental assumptions, we next introduce the notion of a security rationale.

3 Security Rationales

To address a security requirement RQ in an environment $\mathcal{E}$, we deploy a system in $\mathcal{E}$. The system's design, construction, and analysis are guided by a system specification SP. Moreover, deploying the system contributes to RQ if an environmental assumption EA holds true. Reducing RQ to SP and EA must clearly be justified: not every combination of SP and EA contributes to satisfying RQ. Security rationales embody such justifications. A **security rationale** for the four tuple $(\mathcal{E}, RQ, SP, EA)$ is a justification for the following condition: for any system $\mathcal{S}$ and adversary $\mathcal{A}$,

$$\mathcal{S} \models SP \ \wedge \ \mathcal{S} \| \mathcal{E} \| \mathcal{A} \models EA \quad \rightarrow \quad \mathcal{S} \| \mathcal{E} \| \mathcal{A} \models RQ \,. \tag{†}$$

Here $\models$ and $\|$ represent satisfaction and composition. Moreover, SP, EA and RQ can each consist of multiple conjuncts, as illustrated subsequently in Example 5.

Intuitively, a rationale for $(\mathcal{E}, RQ, SP, EA)$ explains that, to address the requirement RQ in the environment $\mathcal{E}$ in the presence of the adversary $\mathcal{A}$, it suffices to deploy the system $\mathcal{S}$ provided that $\mathcal{S} \models SP$ and $\mathcal{S} \| \mathcal{E} \| \mathcal{A} \models EA$. The condition (†) can be used from right to left to **reduce** a security requirement into a system specification and an environmental assumption sufficient for its establishment; see Figure 1.

In (†), the premise $\mathcal{S} \models SP$ guides the design, construction, and analysis of systems, as mentioned above. The inclusion of the system $\mathcal{S}$ in the premise $\mathcal{S} \| \mathcal{E} \| \mathcal{A} \models EA$, which concerns the environmental assumptions, may appear counterintuitive. The adversary's role clarifies this point. The specification SP regulates the system's behaviors over its nominal channels. The adversary, against whom the requirement RQ must hold, may however interact with the system over side channels, i.e. channels not anticipated by SP. The system's interactions with the adversary over these channels must therefore be constrained as well. The environmental assumption EA includes such constraints.

In the following, we further clarify the condition (†). First, there are numerous frameworks for formalizing, verifying, and testing the relation $\mathcal{S} \models SP$ in a precise

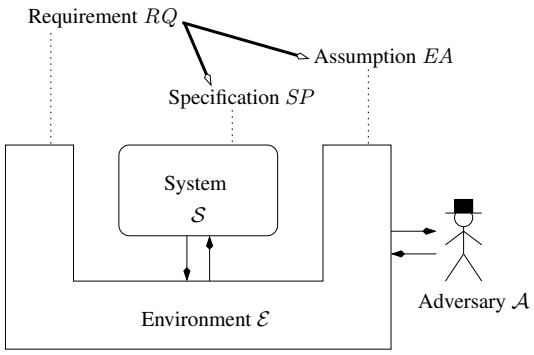

Fig. 1. A security rationale reduces (thick arrows) a security requirement to a specification and an environmental assumption. The validity of the environmental assumption depends on the adversary's capabilities. The adversary interacts (thin arrows) with the system over the environment.

manner. The two other relations in (†) cannot however be readily formalized. In particular, the nebulous entities $\mathcal{E}$ and $\mathcal{A}$ often have no clear boundaries. This poses a major challenge to formalizing the notion of a security rationale. For the rest of this paper, we therefore treat the condition (†) as an informal guideline and as a way to classify verification and refutation objectives.

Second, environmental assumptions and requirements have, in essence, the same type. In particular, (†) would be trivially satisfied if EA were RQ. The resulting reduction would however clearly not help with the requirement's analysis. Moreover, whether a statement is seen as a requirement or an assumption depends on the task at hand. For instance, in Example 3, the assumption that one cannot enter the lab through its window constitutes a requirement if we are interested in constructing the lab building. To satisfy this requirement we may, for example, install window bars; this would be preceded by a specification that would fix the window bars' construction in a way that is deemed sufficient to resist a given adversary.

Third, in the security literature, the environment is sometimes conflated with the adversary. To denote such an adversarial environment, let $\mathcal{E}^* = \mathcal{E}\|\mathcal{A}$. Then (†) boils down to $\mathcal{S} \models SP \ \wedge \ \mathcal{S}\|\mathcal{E}^* \models EA \ \rightarrow \ \mathcal{S}\|\mathcal{E}^* \models RQ$.

Finally, note that any security rationale can account for only a small set of entities and their interactions: we cannot reason about everything in the world. Therefore, any rationale inevitably relies upon the assumption that the excluded entities and interactions play no role in the requirement's satisfaction. This assumption in effect excludes certain adversarial actions. A prominent example is the assumption that the system has no side channels for communicating with the adversary; otherwise, its protection mechanisms can potentially be subverted. This further explains why we cannot dispense with $\mathcal{S}$ in $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models EA$ above.

The following example illustrates the above notions.

Example 5. Consider the R&D laboratory of Example 1. The requirement RQ states that only staff members may enter the lab. The lab has a door that is controlled by an

electronic lock system. We reduce RQ to the requirement (SRQ): the lock opens the door only after a valid key is presented to it. The reduction relies on the following three environmental assumptions. (EA_1) Only staff members have a valid key. (EA_2) The door opens only after receiving the lock's signal.[1] (EA_3) The only way to enter the lab is through the door. Laws of logic justify the reduction.

$$\begin{array}{ll} (EA_1) & \text{hasValidKey}(X) \rightarrow \text{isStaff}(X) \\ (EA_2) & \text{doorOpensFor}(X) \rightarrow \text{signalFor}(X) \\ (EA_3) & \text{enterLab}(X) \rightarrow \text{doorOpensFor}(X) \\ (SRQ) & \text{signalFor}(X) \rightarrow \text{hasValidKey}(X) \\ \hline (RQ) & \text{enterLab}(X) \rightarrow \text{isStaff}(X) \end{array} \qquad (\star)$$

The assumptions constrain the adversary's capabilities. The assumption EA_1, for instance, excludes numerous adversarial actions, both simple and elaborate. For example, according to EA_1, an adversary is not capable of bribing staff members to obtain a valid key. Similarly, the adversary cannot forge a valid key. The excluded adversarial actions clearly cannot be feasibly enumerated.

In the final step, we reduce the requirement SRQ to the following specification for the lock system's nominal communication channel: (SP) the output signal `open` is produced only after receiving as input a `key` that belongs to the set `validKeys`. This reduction is justified by two assumptions EA_I and EA_S. The assumption EA_I states that the set `validKeys`, the input `key`, and the signal `open` are interpreted as expected, and that an entity cannot send a key to the lock system unless the entity has the key. The latter conjunct intuitively bridges the gap between the predicate `hasValidKey(X)` and the key the lock system receives from an entity `X`. The assumption EA_S states that all the communication channels between the lock system and the adversary are regulated by SP. It excludes for instance the possibility that the lock system has a hidden backdoor that bypasses its functionalities, or that disrupting the lock's electricity supply (which constitutes an "input" to the lock system) would leave the door open.

The arguments above constitute a security rationale for the tuple $(\mathcal{E}, RQ, SP, EA)$, where $\mathcal{E}$ is the lab's environment, RQ and SP are defined above, and EA is the conjunction of the assumptions EA_1, EA_2, EA_3, EA_I, and EA_S. Note that EA_I and EA_S cannot be expressed as assumptions on the environment alone: the lock system must be considered too. $\triangle$

The reduction steps carried out in a security rationale can be graphically represented as a **reduction tree**. Formally, a reduction tree is simply an and-or tree where the root denotes a security requirement, and the leaves are system specifications and environmental assumptions; see Figure 2.

Security rationales justify a reductive strategy for addressing security requirements. Such justifications can, in part, be formalized in a suitable proof system and justified using laws of logic, as $(\star)$ suggests. Laws of physics, such as nothing travels faster than the speed of light, can also be part of a security rationale. Formal models of the problem domain can assist security engineers with this task; cf. [1, 5, 14].

[1] We will abstract away from further temporal aspects in this example. For instance, once the door has been closed, it remains closed until the next signal arrives, and only one person can pass through the door while it is open.

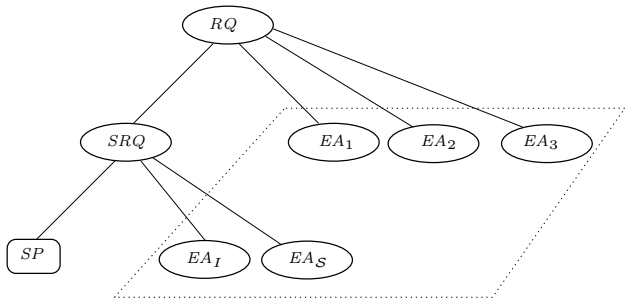

Fig. 2. A reduction tree for Example 5. Requirements and environmental assumptions are depicted as ovals, while specifications are depicted as rectangles. The dotted polygon contains the environmental assumptions. All branches here are and-branches.

4 Security Cases

In this section, we introduce the notion of a security case. Intuitively, a security case explains why a rationale for a given security requirement is applicable to a concrete system in the presence of a specific adversary. Suppose we have a security rationale for the tuple $(\mathcal{E}, RQ, SP, EA)$. Then, deploying a system $\mathcal{S}$ in the environment $\mathcal{E}$ guarantees that RQ holds in the presence of an adversary $\mathcal{A}$ if the following condition holds:

$$\mathcal{S} \models SP \quad \wedge \quad \mathcal{S} \| \mathcal{E} \| \mathcal{A} \models EA\,. \tag{‡}$$

This statement is a direct consequence of (†). A **security case** is an argument for (‡)'s truth, for a concrete system $\mathcal{S}$ and a specific adversary $\mathcal{A}$. If the rationale's reduction steps are represented as a reduction tree, then a security case is an argument for the satisfaction of the tree's leaves.

Three remarks are due here. First, security cases are analogous both to safety cases, which argue for the safety of, say, vehicles (see for example ISO 26262-1:2011), and to dependability cases [12]. Security cases are (ideally) provided by security designers and analysts who explain why deploying $\mathcal{S}$ in the environment $\mathcal{E}$ solves the problem of addressing RQ in the presence of the adversary $\mathcal{A}$. For example, software verification techniques that demonstrate that a software system $\mathcal{S}$ satisfies a specification SP can contribute to a security case.

Second, the adversary's capabilities do not enter into a security rationale itself. Instead, once a specific adversary has been identified, for example, through risk analysis, the security case is given to justify the environmental assumptions' validity in the adversary's presence. The following example illustrates this point.

Example 6. Consider the security rationale of Example 5. This rationale does not depend on any particular adversary or system. However, the validity of the environmental assumptions critically depends on the adversary's capabilities, and the validity of the specification depends on the system's behaviors. For instance, the assumption EA_1 is violated if the adversary can threaten or bribe a staff member and thereby obtain a valid

key. A security case here must argue that the given adversary, say, curious visitors, cannot violate this assumption. Similarly, the security case explains why a given lock system's behaviors over its nominal channels satisfy SP. △

Third, whether or not a system satisfies a specification does not depend on the adversary's capabilities, as is evident in the condition (‡). This is a central point: systems can be designed, developed, and evaluated without knowledge about the environment where they will be deployed. That a system contributes to the security of protected resources in a given adversarial environment must be justified using security cases. This observation may seem counterintuitive as, for example, buffer overflow attacks and SQL injections, where an adversary takes control of a system by providing it with "malicious" inputs, are prevalent. We remark that these attacks exploit a system's inadequate handling of malformed inputs. They can therefore be addressed by providing an adequate specification for the system's interface and requiring that the system satisfies it.

As mentioned in §3, the environmental assumptions always exclude certain adversarial actions. These exclusions cannot be justified without accounting for all interactions in the world, which is clearly infeasible. Therefore, to construct a manageable model of the environment, security cases invariably depend on **closed-world assumptions**, stating that what has not been considered plays no role in satisfying the given security requirement. Closed-world assumptions thus complete security cases in this merely formal sense [25]; see also Simon's *empty world hypothesis* [26]. The following example illustrates this point.

Example 7. Consider the security rationale of Example 5. The validity of $EA_{\mathcal{S}}$, which states that the system has no side channels, depends on the adversary's capabilities and the system's behaviors. Suppose the lock system leaves the door open if its power is disrupted. The assumption $EA_{\mathcal{S}}$ is then not valid in the presence of an adversary who can cut off the system's power. It might however be valid for a weaker adversary. A security case here explains why a given system and adversary cannot communicate over this particular side channel in the environment $\mathcal{E}$. Alternatively, if the system leaves the door locked when the power is disrupted, then the security case can argue that although an adversary can affect the system over this side channel, the result does not adversely affect RQ's satisfaction. To complete the argument for $EA_{\mathcal{S}}$'s validity, all possible channels should be considered. These, however, cannot all be enumerated and argued for. The security case must therefore ultimately rely on the closed-world assumption that the considered side channels are the only ones relevant for RQ's satisfaction. △

5 Security Testing

In this section, we define functional testing and security testing, and clarify their relationship. We then introduce two types of security tests, and illustrate them through examples from practice. Finally, we discuss vulnerabilities and their remediation, associated with these two test types.

By **functional testing** we refer to any process aimed at refuting the hypothesis that a system satisfies its (functional, security, or other) specifications. That is, given a system $\mathcal{S}$ and a specification SP, functional testing aims at refuting the hypothesis $\mathcal{S} \models$

SP. Here we do not distinguish between black-box and white-box analysis. By **security testing** we refer to any process aimed at refuting the hypothesis $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models RQ$, for a system $\mathcal{S}$, environment $\mathcal{E}$, adversary $\mathcal{A}$, and security requirement RQ. Note that the purpose of both types of testing is to refute a hypothesis, rather than to verify it. This understanding, which is well-established in the literature [9, 21], sharply separates constructing security cases from security testing.

We remark that our notion of functional testing is more general than the term's conventional denotation in the literature, e.g., [2, 4, 21]. This is simply because, in our theory, a specification need not be confined to a system's desired functions, distilled, say, from its use cases. A bound on the system's delay in producing outputs, as well as a threshold on the system's electromagnetic radiation level are examples of system specifications. Tests aiming to refute these specifications therefore constitute functional tests in our theory. Conventionally, they are usually not deemed as functional tests because a system's delays and radiation levels are typically not considered to be part of a system's functionality; see also [10] for the murky boundary between functional and non-functional specifications. In our theory, the essence of a functional test is that it applies to the system's communication channels that are described in and constrained by the system's specification. To avoid confusion, we refer to the conventional forms of functional tests as **restricted functional tests**.

We now turn to security testing. Suppose that, in an environment $\mathcal{E}$, a requirement RQ is intended to be satisfied based on a rationale for $(\mathcal{E}, RQ, SP, EA)$. Let $\mathcal{S}$ be a system deployed in $\mathcal{E}$ that is intended to satisfy (‡), in the presence of an adversary $\mathcal{A}$. Perhaps surprisingly, refuting either conjunct of (‡) does *not* entail refuting $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models RQ$, which is the objective of security testing. However, the refutation of $\mathcal{S} \models SP$ or $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models EA$ does, of course, demonstrate that the intended rationale's premises are false for the system $\mathcal{S}$ and the adversary $\mathcal{A}$: the condition (†) is true due to the failure of its antecedent and one cannot construct a security case here. Therefore, the refutation of one of (‡)'s conjuncts *suggests* that the requirement RQ is violated because it is unlikely that RQ is satisfied due to unintended causes. This observation motivates the following hypothesis: If $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models RQ$, for a system $\mathcal{S}$ and an adversary $\mathcal{A}$, then $\mathcal{S} \models SP$ and $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models EA$. We call this the **intentional security hypothesis**, in short **H**.

Intuitively, **H** states that a security requirement is never satisfied unintentionally: a system addresses a security requirement by design, not by accident. Note that the hypothesis amounts to the condition (†)'s converse. This is expected: the condition (†) supports constructing security cases for verifying a security requirement's satisfaction. Security testing, whose goal is to refute the requirement's satisfaction, must rely on (†)'s converse, namely **H**. We show in §5.2 that **H** has been tacitly assumed in the literature.

5.1 S-Tests and E-Tests

Based on **H**, the tester can refute the hypothesis $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models RQ$ by refuting one of (‡)'s conjuncts. This results in the following two types of security tests.

S-Tests: Test the system with respect to its specification.

Fig. 3. S-tests, whose purpose is to refute the hypothesis that a system satisfies its specification, include restricted functional tests, which apply to the functionalities the system must offer. E-tests, in contrast, attempt to violate an environmental assumption in an adversarial environment.

A test of this type, called an S-test, is intended to refute $\mathcal{S} \models SP$, which is an instance of functional testing. Tools and techniques for generating and automating functional tests can therefore be used here; see for example [2, 4]. Note that S-tests pertain to symbol manipulating entities and are therefore independent of the adversary. Moreover, restricted functional tests are instances of S-tests. For example, suppose a radio transmission system must satisfy the specification that transmitted messages should be encrypted with 1024-bit keys. Restricted functional tests can be applied to this system because the specification describes a use case of the system.

E-Tests: Test the validity of the environmental assumptions.

A test of this type, called an E-test, is intended to refute the hypothesis $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models EA$. Refuting this hypothesis is what distinguishes security testing from functional testing. Namely, functional tests pertain to a system's behaviors over its interface, described by a specification. In contrast, security E-tests apply not only to systems but also to a nebulous environment and an adversary with no interface (see §2). Therefore, testing the validity of environmental assumptions cannot be reduced to providing an input and observing an output over a definite interface. These tests are therefore not an instance of functional tests: they pertain to actual entities in the world. In particular, they depend on the adversary's capabilities. The diagram of Figure 3 illustrates the relationship between these two types of tests.

Example 8. Consider the scenario of Example 5, with the reduction tree depicted in Figure 2 for the requirement RQ. The purpose of security testing is to refute the hypothesis that RQ is satisfied in the presence of a given adversary $\mathcal{A}$. As previously explained, refuting the validity of the reduction tree's leaves (which is the goal of E-tests and S-tests) does *not* entail that RQ is violated, because RQ can be satisfied due to unanticipated reasons. It is only by $\mathbf{H}$ that design errors imply RQ's violation. We consider the task of violating some of Figure 2's leaves in the following.

To violate the leaf SP, the tester tries to refute the hypothesis that the lock system satisfies the specification SP. The tester may, for instance, input very large keys into the lock system, where a key is a sequence of bits. If a buffer overflow is discovered, then the adversary might be able to take control of the lock and produce an `open` signal without possessing a valid key. Note that to violate SP the tester need not elicit the adversary's capabilities. The lock system must satisfy SP on its nominal channel for all possible inputs and outputs. This is an S-test. In contrast, the tests below are E-tests.

To violate EA_I, the tester checks if the lock system's local variables are misinterpreted, for example, the set `validKeys` might not actually consist of valid keys. If a staff member leaves the R&D team, then his key might still be stored in `validKeys`. The tester also checks whether the lock system is susceptible to replay attacks. If so, then EA_I is violated because the adversary can simply record the interaction between a valid key and the lock system and later send a valid key to the lock without legitimate possession of the key. Whether these scenarios refute EA_I's validity in the presence of a given adversary clearly depends on the adversary's capabilities.

To violate EA_2, the tester may try to intercept the communication between the lock and the door to inject an `open` signal. The tester may also assess the feasibility of breaking, or unhinging, the lab's door. To violate EA_3, the tester may try climbing through the window. The feasibility of these attacks naturally depends on the environment and the adversary's capabilities. If, for instance, the window is barred and the adversary neither has a metal saw nor is capable of squeezing through the bars, then climbing in through the window is infeasible, indicating that EA_3 is not refuted in these scenarios. △

As the above example illustrates, when testing environmental assumptions and requirements, the tester must take the adversary's capabilities into account. For each goal the tester may ask whether the adversary can achieve it. Specific goals, such as unhinging a door, lead to specific questions regarding the adversary's capabilities. General goals, such as violating the assumption EA_2, which excludes a wide range of adversarial actions, lead to generic questions that cannot be directly answered. The tester must then elicit a list of attack scenarios and determine whether the adversary can realize them. This list can be developed by brainstorming and using experience with similar requirements. This can also be aided by consulting sources like [7], which go beyond enumerating common system vulnerabilities and consider malicious interaction from the environment. The investigated scenarios will however never be complete, because accounting for all possible interactions in the world is infeasible. Security testing is therefore an open-ended processes, hence inherently incomplete. Note that this incompleteness is orthogonal to the incompleteness of functional tests, which is rooted in the infinite cardinality of the domains where test inputs are selected. The difference is that in functional testing one picks inputs from a delimited, albeit infinite, domain, whereas E-tests come from a domain with no boundaries.

The following example illustrates the essentially unlimited creativity required by a security tester to anticipate all possible attack scenarios.

Example 9. A British secret operation, known as the Four Square Laundry affair, was carried out in Northern Ireland to collect information about the residents of a troubled neighborhood [20]. A rogue laundry service van visited the neighborhood regularly, and sent the collected laundry for various tests and inspections before washing it. The tests included checking for traces of explosive material or blood. The service also noted changes in the amount or kinds of clothing sent by each household for washing, which could indicate the presence of guests, and so forth. △

The separation between S-tests and E-tests explains why security testing is harder than functional testing. A system specification describes the system's behaviors over its interface. It can therefore be used to construct functional tests, for example S-tests, independently from the adversary's capabilities and the environment in which the system

is deployed. When it comes to security testing, the tester must also check the validity of requirements in an adversarial environment. Environments and adversaries are nebulous entities, with no clear interface. How, say, an environmental assumption can be violated depends on the adversary's capabilities, the environment's properties, and the system's behaviors. E-tests for checking an assumption's validity are only as thorough as the attack scenarios the tester anticipates.

5.2 S-Tests and E-Tests in Practice

Applying security testing in practice is challenging. If the security case (or the security rationale) intended to guarantee a resource's security is unavailable, then the tester must reconstruct, or approximate, it. This includes eliciting the adversary's capabilities and explicating specifications and environmental assumptions. These tasks are notoriously hard in practice; see for example [14, 30]. Even when the security case and the security rationale are available, security E-tests amount to anticipating how the adversary can invalidate an environmental assumption or a requirement. This task defies prescriptive guidelines such as those available for functional testing. The effectiveness of E-tests therefore depends largely on the tester's creativity and resources; see the Four Square Laundry example above.

These observations imply that security testing is largely a manual task that defies specific, thorough guidelines. It is therefore not surprising that existing methods fall short when it comes to E-tests. Below, we substantiate this claim by showing that existing security testing techniques have little to say in this regard.

Risk-Based Security Testing. Risk-based security testing [18, 19, 24] starts by explicating system specifications from risk analysis, misuse case diagrams, and other design and analysis documents. Roughly speaking, a risk corresponds to a security requirement that demands the risk's mitigation. The countermeasure that is intended to reduce or eliminate the risk can then be seen as a specification that defines how a system must implement the mechanisms that address the corresponding requirement. Afterward, risk-based security testing reduces security testing to S-tests applied to the mitigation mechanisms. E-tests are absent here because the environmental assumptions and the adversary that would make up a corresponding security rationale are not identified.

Fuzz Testing and Fault Injection. Fuzz testing [11,27] and fault injection techniques [29] aim at refuting generic system specifications such as: the system does not access unallocated memory areas. That is, they refute $\mathcal{S} \models SP_g$, for generic specifications SP_g and they therefore amount to S-tests. These techniques can be seen as generating S-tests guided by security-relevant fault models. For example, programs often fail to check their inputs length or format, and they have inadequate exception handling when dependency relations fail. Such fault models reflect how an adversarial environment may interact with the system. Consequently, they give rise to tests that are tailored to violate security-relevant specifications. E-tests are nonetheless absent here, simply because the resulting tests pertain to a system's nominal channels only; they do not analyze side channels and environmental assumptions.

Vulnerability-Driven Security Testing. Tests that try to identify a known, anticipated vulnerability in a particular system are sometimes said to be driven by that vulnerability. Since these tests are concerned with systems, they are clearly S-tests. OWASP's security test patterns fall under this class of security tests [24].

A more elaborate example of vulnerability-driven security testing is the NIST proposal [23] that associates security tests with security features of cryptographic modules. An example is that "environmental failure protection [...] features shall protect the cryptographic module against unusual environmental conditions or fluctuations (accidental or induced) outside of the module's normal operating range that can compromise the security of the module" [23]. The document associates a number of tests to this security feature, including "the tester shall extend the temperature and voltage outside of the specified normal range and determine that the module either shuts down to prevent further operations or zeroizes all plaintext secret and private keys and other unprotected [critical security parameters]". The NIST proposal is helpful in explicating how a module should behave in abnormal conditions, but it cannot describe under which assumptions on the adversary's capabilities and the environment a security requirement can be translated into the specifications subjected to functional tests. Note that although the NIST's suggested tests are not instances of restricted functional tests, they nevertheless apply to a system's communication channel that has been regulated by the system's specifications. They are therefore S-tests. E-tests are absent here as well.

In short, existing security testing methods and tools ignore E-tests. Since they all address security specifications, they tacitly assume that if the system violates its specification, then the security requirement is also violated. This amounts to the intentional security hypothesis, introduced in §5, about which the literature has not been explicit. The aforementioned shortcomings should not be construed as a criticism of the existing techniques' value. Rather, our security test types should be seen as a tool for delimiting their scope and reach. As mentioned before, E-tests depend on the adversary and target closed-world environmental assumptions that are impossible to delimit. It is therefore not surprising that, in contrast to S-tests, E-tests do not admit automation.

We conclude this section with two remarks. First, adversary models themselves are not subjected to E-tests (or S-tests). For example, discovering that a safe can be opened using standard office equipments demonstrates that the assumption that a curious co-worker could not open the safe has been false all along. It however does not help us decide whether a curious co-worker is a suitable adversary model for the documents protected by the safe. In general, E-tests and S-tests do not account for flaws rooted in unelicited requirements or weak attacker models. Requirements and the adversary are the parameters with respect to which these test types are defined. They are not themselves subject to these tests.

Second, the observations above shed light on the notion of adequacy for security tests. It is immediate that the adequacy of S-tests can be defined based on functional adequacy measures, such as coverage [31] and mutation analysis [8], and security-specific metrics such as [28]. The adequacy of E-tests, however, is an entirely different matter. Ideally, the validity of each environmental assumption must be "adequately" tested. These assumptions are however not only hard to explicate, but their validity also relies upon closed-world assumptions that can never be thoroughly tested. No finite set

of security tests can therefore constitute an adequate set of E-tests. We return to this conundrum in §6.

5.3 Vulnerability Remediation

We can classify security vulnerabilities based on our test types. Let $\mathcal{S}$ be a system, $\mathcal{E}$ an environment, $\mathcal{A}$ an adversary, and RQ a security requirement. By a security **vulnerability** we refer to any cause for the violation of the security requirement, i.e., the violation of $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models RQ$. Clearly this notion of a vulnerability is more general than, say, programming flaws.

We introduce two classes of vulnerabilities: S-vulnerabilities, and E-vulnerabilities. **S-vulnerabilities** are those vulnerabilities in the system $\mathcal{S}$ that lead to a violation of its specification SP. Due to **H**, these are indeed vulnerabilities as they lead to a violation of RQ. These vulnerabilities are revealed through S-tests, and remediating them amounts to fixing the system. **E-vulnerabilities** are those vulnerabilities that invalidate the environmental assumption EA. That is, vulnerabilities in this class cause the relation $\mathcal{S}\|\mathcal{E}\|\mathcal{A} \models EA$ to fail. These too are vulnerabilities due to **H**, as they lead to a violation of RQ. To remediate an E-vulnerability, fixing the system alone is insufficient. The system must be re-engineered and the security rationale must be updated.

After fixing a system to address an S-vulnerability, only the system must be analyzed using S-tests; carrying out E-tests is unnecessary. Moreover, since the system's specification has not changed, these S-tests can be seen as regression tests. However, after re-engineering the design and updating the security rationale to address an E-vulnerability, both S-tests and E-tests must, in general, be carried out to analyze the security of the new design. Since these tests must address the new design's specification and environmental assumptions, they cannot be seen as simple regression tests. The following example illustrates these classes.

Example 10. Consider the scenario of Example 5, analyzed in part in Example 8. A window through which the adversary can enter the office is an E-vulnerability. To address it, new systems, such as window bars, can be installed in the environment. This system must then be tested with respect to its specification. As a second example, suppose that former staff members still have keys that are accepted by the lock system. This causes the environmental assumption EA_1 to fail. To address this E-vulnerability, the specification SP must be extended with the specification of a suitable key revocation mechanism. This likely entails changing the lock system entirely, installing a key revocation server, and so forth. These systems must then be tested with respect to the extended specification. △

We have ignored flaws due to changes in the requirements or a mismatch between the stake-holder's expectations and the requirements. Although such cases are common in practice [15], they fall outside this paper's scope.

6 Concluding Remarks

Starting with the fundamental distinction between a system specification and a security requirement, we have provided a simple theory of security testing. Its ingredients — se-

curity rationales, security cases, the intentional security hypothesis, S-tests and E-tests — provide a basis for explaining the verification and refutation of security requirements in general, and security testing in particular. Our theory highlights the limitations of many testing and other quality assurance methods for reasoning about the security of systems: the vast majority of methods target the relationship between systems and their specifications, but not the assumptions made on their environments.

Targeting environmental assumptions is hard. One must ultimately resort to a closed-world assumption and posit that the adversary can only interact with the system and the environment in limited ways. As a result, the set of possible counter-examples is not only infinite, its domain cannot be precisely delimited. Hence, E-tests, which target environmental assumptions, defy automation and systematization.

The above difficulties raise the question of how practitioners can best approach E-testing and judge the quality of the resulting E-tests. We do not have the answer to this question. And any answer will certainly not be in terms of a logical method or formalism with conventional notions of completeness or coverage. Since testers' creativity and experience play a central role in refuting environmental assumptions, there is value in studying and learning from attacks [3, 6]. We believe our theory can help in this regard as it suggests a frame of reference for documenting, classifying, and reusing the knowledge obtained through such studies. This includes explicating the assumptions that have been violated, associating common assumptions with attacks, and exploring possibilities for generalizations. Moreover, threats on different classes of systems and environments can be cataloged along with countermeasures; see, e.g., [7]. These catalogs can be analyzed using this frame of reference, highlighting cases where the attacks and mitigation methods refer to assumptions or specifications that are left implicit. Making these explicit can contribute to the body of knowledge developed around E-tests.

Security testing requires an open mind and a vivid imagination. It goes far beyond the well-charted territory of functional tests. One must raise one's sights to look beyond the machine and target the world as well.

Acknowledgment. We thank Peter Müller and Petar Tsankov for their comments on this paper.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
2. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
3. David A. Basin and Srdjan Capkun. The research value of publishing attacks. *Commun. ACM*, 55(11):22–24, 2012.
4. Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition, 1990.
5. Dines Bjorner. *Software Engineering 3: Domains, Requirements, and Software Design (Texts in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.
6. BSI. A penetration testing model, 2003. The German Federal Office for Information Security.
7. BSI. IT Grundschutz Kataloge, 2014 (Version: 14). The German Federal Office for Information Security.

8. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
9. Edsger W. Dijkstra. Notes on structured programming. Technical Report T.H. Report 70-WSK-03, Technological University Eindhoven, April 1970.
10. Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference, RE*, pages 21–26. IEEE Computer Society, 2007.
11. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *ACM Queue*, 10(1):20, 2012.
12. Daniel Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, 2009.
13. Michael Jackson. The world and the machine. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 283–292, New York, NY, USA, 1995. ACM.
14. Michael Jackson. *Problem Frames*. Addison-Wesley, 2001.
15. Ann Johnson. *Hitting the Brakes: Engineering Design and the Production of Knowledge*. Duke University Press, 2009.
16. Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA*, pages 361–372. IEEE Computer Society, 2014.
17. R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. *SIGOPS Oper. Syst. Rev.*, 9(5):132–140, November 1975.
18. Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.
19. C.C. Michael, Ken van Wyk, and Will Radosevich. Risk-based and functional security testing, Last revised: July 05, 2013. `https://buildsecurityin.us-cert.gov/`.
20. Ed Moloney. *A Secret History of IRA*. Penguin Canada, 2003.
21. Glenford Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley, 3 edition, 2011.
22. Ruth Nelson. What is a secret - and - what does that have to do with computer security? In *Proceedings of the Workshop on New Security Paradigms*, pages 74–79. IEEE, 1994.
23. Derived test requirements for FIPS PUB 140-2, security requirements for cryptographic modules, 2011. NIST, CSEC and CMVP Laboratories Draft.
24. OWASP. Testing guide v. 4, Accessed on 9/3/2014. `https://www.owasp.org`.
25. Raymond Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.
26. Herbert A. Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482, 1962.
27. Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.
28. Petar Tsankov, Mohammad Torabi Dashti, and David A. Basin. Semi-valid input coverage for fuzz testing. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 56–66. ACM, 2013.
29. Jeffrey Voas and Gary McGraw. *Software Fault Injection*. Wiley, 1998.
30. Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *Proceedings of the 22nd USENIX Conference on Security*, pages 399–414, 2013.
31. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.