Semi-valid Input Coverage for Fuzz Testing

Petar Tsankov, Mohammad Torabi Dashti and David Basin
Institute of Information Security
ETH Zurich, Switzerland
{ptsankov, torabidm, basin}@inf.ethz.ch

ABSTRACT

We define semi-valid input coverage (SVCov), the first coverage criterion for fuzz testing. Our criterion is applicable whenever the valid inputs can be defined by a finite set of constraints. SVCov measures to what extent the tests cover the domain of semi-valid inputs, where an input is semi-valid if and only if it satisfies all the constraints but one.

We demonstrate SVCov's practical value in a case study on fuzz testing the Internet Key Exchange protocol (IKE). Our study shows that it is feasible to precisely define and efficiently measure SVCov. Moreover, SVCov provides essential information for improving the effectiveness of fuzz testing and enhancing fuzz-testing tools and libraries. In particular, by increasing coverage under SVCov, we have discovered a previously unknown vulnerability in a mature IKE implementation.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Security, Reliability

Keywords

fuzz testing, coverage criteria, security testing

1. INTRODUCTION

Coverage criteria are an integral part of testing practice [12] and quality assurance standards [26]. Measuring a test set's coverage with respect to a given criterion is relevant for exposing failures in the system under test (SUT): low coverage hints that tests are missing and suggests how to improve the test set. Existing coverage criteria are however ill-suited for measuring the coverage of fuzz testing. In particular, they do not measure what fuzz testing is all about, namely executing the SUT with *semi-valid* inputs.

In this paper, we propose a coverage criterion for fuzz testing. Before explaining why a coverage criterion specific to fuzz testing is needed, and what the characteristics of our proposed criterion are, we briefly recall the principles of fuzz testing. There is no formal definition of fuzz testing, and indeed the boundary between fuzz testing and other testing techniques is not sharp. Fuzz testing is however commonly understood as executing the SUT with inputs that are not foreseen by the SUT's specification. The SUT is then checked for the presence of generally undesired behaviors, such as memory access violations; see [19, 32].

The inputs used for fuzz testing must be semi-valid [22], i.e. inputs that are not valid according to the specification, but not entirely invalid either. This is admittedly a vague definition. The point is that most often entirely-invalid inputs are filtered out by input parsers, and therefore do not exercise the software beyond the parser code. For instance, a program intended to work with `jpeg` files is unlikely to accept a `pdf` file. The program is however likely to accept modified `jpeg` files where a bit has been flipped, and in the course of its execution throw an exception. In this example, the modified `jpeg` file is a semi-valid input, whereas `pdf` files are not. In this paper, we build upon the notion of semi-valid inputs to define a coverage criterion for fuzz testing.

We define the *semi-valid input coverage* criterion, abbreviated as SVCov. Our criterion is applicable whenever the valid inputs can be defined by a finite set of constraints. For instance, the valid inputs for a software implementing a communication protocol are exactly those that satisfy all the constraints defined in the protocol's specification. We define a semi-valid input as any input that satisfies all the constraints but one. A test set T is deemed thorough according to SVCov if and only if, for each constraint c, the set T contains at least one test that violates c and simultaneously satisfies all the other constraints.

We remark that a myriad of coverage criteria for testing have been defined and evaluated in the literature, see e.g. [7, 36]. Many of the existing criteria are applicable to fuzz testing too. For example, there is empirical evidence that increasing statement coverage increases the effectiveness of fuzz testing in terms of discovering faults [32]. The existing criteria measure the coverage of a set of fuzz tests in terms of the number of statements executed by the tests, etc. They do not measure, however, to what extent the SUT is exercised with semi-valid inputs. A test set that contains only valid inputs may, for example, execute all the code statements in the SUT; nevertheless, testing the SUT with such a test set is clearly not an instance of thorough fuzz testing. In contrast to the existing criteria, SVCov measures to what extent tests cover the domain of semi-valid inputs and hence reflects the definition of fuzz testing more closely than other criteria. We do not see SVCov as a substitute for the existing criteria. Instead, as with any other software metric, SVCov complements and balances other applicable criteria, cf. [5].

Contributions. We propose SVCov, the first coverage criterion for fuzz testing. To show its practical value, we investigate SVCov with respect to three central requirements:

R1 *Feasibility:* One must be able to (1) precisely define the semi-valid inputs of the SUT from real-world specifications in a reasonable amount of time and (2) efficiently measure SVCov.

R2 *Relevance to coverage:* Measuring SVCov must provide the tester with meaningful information on how to improve a test set's coverage.

R3 *Relevance to discovering faults:* Increasing SVCov of a test set results in discovering additional faults in the SUT, if any exists.

To judge whether SVCov satisfies these requirements, we conducted a case study on fuzz testing the Internet Key Exchange protocol (IKE). Fuzz testing security protocols such as IKE is a recognized challenge [32] because security protocols are stateful, they have complex input structures, and they use encryption. The challenging nature of fuzz testing IKE implementations, and the scale of IKE specifications (including three RFCs [25, 17, 13]), make IKE a representative candidate for our study.

Our experiments with IKE confirm that it is indeed feasible to precisely define the set of semi-valid inputs by extracting constraints from real-world protocol standards, which in the case of IKE span a number of RFCs. Moreover, the time needed to check the satisfiability of the constraints for a test (used to measure SVCov) is on average two orders of magnitude smaller than the time needed to execute the test. The overhead of measuring SVCov during testing is therefore negligible.

We measured SVCov and analyzed the results to find out why some semi-valid inputs were missing in the tests. The analysis pointed at a number of subtle problems with our fuzz-testing setup and provided us with essential information for improving the coverage of the tests. Concretely, we discovered (1) implementation bugs in SecFuzz [34], the fuzz-testing tool used in the study, (2) imprecision in SecFuzz's fuzz operators, and (3) redundancy in the constraints that we had extracted from IKE's RFCs. SVCov also pointed at missing valid inputs, which are critical for SecFuzz's effectiveness and, in general, for mutation-based fuzz-testing techniques (further details are given in § 3.4.3). Fixing these problems significantly improved the tests' coverage.

We discovered that the tests that are missing in the first experiment are important; in the second experiment, we found a previously unknown vulnerability in a popular, stable implementation of IKE. The discovered vulnerability is security-relevant: an attacker can exploit the vulnerability to subvert IKE. We have communicated the vulnerability to the software vendor.

Summing up, our case study provides evidence that measuring SVCov is feasible, and it can be used to improve the quality of the fuzz-testing tools and libraries in general and the coverage of test sets in particular. Moreover, increasing SVCov of our test set uncovered a severe previously unknown security vulnerability. Overall, our initial experiences with SVCov are very encouraging; see § 5 for our future work.

A word on terminology. By *testing* we mean executing the SUT and checking its behavior for *failures*. Failures are manifestations of *faults*, which are the result of human errors in constructing the SUT; cf. [14]. *Vulnerabilities* are security-relevant faults, which are exploitable by attackers. Each *test* (also known as *test case*, or *test input*) is a pair of input and expected output. In the context of fuzz testing, test inputs are those that are not foreseen by the SUT's specification. That is, the specification does not prescribe expected outputs for the inputs used for fuzz testing; hence, it cannot be used as a test oracle. Test oracles used in fuzz testing therefore typically check for generally undesired behaviors. In other words, expected outputs for all test inputs are the same, e.g. no memory access violations occur.

The purpose of testing is to discover faults in the SUT. A set of tests is *adequate* if and only if it exposes all the SUT's faults. This definition of adequacy, although to the point, is of little practical value since the entire set of faults in the SUT is not known (otherwise, no testing would be needed). The adequacy of test sets therefore cannot be measured according to this definition. In particular, in the troubling case where a test set reveals no failures, we cannot decide whether the SUT is fault-free, or the test set is simply inadequate.

A set of tests is deemed *thorough* according to a given coverage criterion if and only if it achieve full (100%) coverage. We remark that the relation between thoroughness and adequacy is tenuous: test sets that are not thorough are likely to miss important tests, although not every thorough test set is adequate.

Outline of the paper. The remainder of this paper is organized as follows. In § 2 we formally define SVCov and describe how to instantiate and use it. In § 3 we demonstrate the practical value of SVCov in a case study on fuzz testing IKE. We review related coverage criteria and compare them to SVCov in § 4. Finally, in § 5 we conclude the paper and discuss our future work.

2. SEMI-VALID INPUT COVERAGE

We briefly describe the notion of coverage for testing. Afterwards we introduce SVCov, our criterion for fuzz testing, and we describe how to instantiate and use SVCov.

2.1 Coverage Criteria Axioms

Numerous coverage criteria for testing have been defined and studied in the literature, see e.g. [7, 36, 20]. Coverage criteria are used to measure the progress of testing with respect to the criterion at hand and to decide when testing can be stopped or, occasionally, to automatically generate and prune test sets. One however must be careful not to over-fit tests to coverage criteria as this may result in less effective tests [27, 30].

A coverage criterion must satisfy a number of basic axioms: the empty test set must be assigned zero coverage and a test set's coverage must never decrease when the test set is enlarged. Furthermore, to exclude the trivial criterion that uniformly assigns zero to every test set, one requires that for any coverage criterion there is at least one thorough test set. These requirements are often stated by the axioms of *inadequacy of the empty set*, *monotonicity*, and *applicability* in the literature; see e.g. [35, 36].

Fuzz testing exercises the SUT with semi-valid inputs. We therefore add the following axiom for coverage criteria for fuzz testing:

Fuzz-testing coverage axiom: A test set that only consists of valid inputs achieves zero coverage.

This axiom states that a test set does not fuzz-test the SUT unless invalid inputs belong to the set. Any test set containing only valid inputs must therefore be assigned zero coverage when it comes to fuzz testing.

It is easy to check that various coverage criteria based on the SUT's model and program structure (such as statement coverage, branch coverage, transition coverage, decision coverage, etc.) fail to conform to the fuzz-testing coverage axiom. Similarly, coverage criteria based on input domain partitions and their boundary values do not explicitly distinguish between valid, semi-valid, and entirely-invalid inputs (we define these terms shortly). They also do not conform to the fuzz-testing coverage axiom. This is not surprising as the existing coverage criteria are not tailored to fuzz testing. Below, we introduce a coverage criterion for fuzz testing that conforms to the fuzz-testing coverage axiom.

2.2 SVCov Coverage

We define a coverage criterion for fuzz testing. The definitions in this section are abstract: we do not specify the input domains, how constraints are characterized, etc. In § 3, we give concrete examples of all the concepts defined here.

Let $\mathcal{I}$ be the infinite set of all possible inputs. We do not further specify $\mathcal{I}$: inputs can be character strings, sequences of communication messages, `jpeg` files, etc. An *input constraint* c is a subset of the possible inputs, i.e. $c \subseteq \mathcal{I}$. An input i *satisfies* the constraint c iff $i \in c$; otherwise, we say that i *violates* c. We assume that a finite nonempty set S of constraints is given such that S defines the set of valid inputs for the SUT at hand. The set of valid inputs, denoted I_{valid}, is then the largest set of inputs that satisfies all the constraints in S. Formally,

$$I_{\mathsf{valid}} = \bigcap_{c \in S} c \ .$$

Input i is an *invalid input* iff $i \notin I_{\mathsf{valid}}$. An invalid input therefore violates at least one constraint in S. Invalid inputs that violate exactly one constraint are called *semi-valid* inputs. Semi-valid inputs are considered to be particularly effective for fuzz testing because they are likely to pass through the SUT's input validation filters [32]. We discuss alternative definitions for semi-valid inputs in § 5.

To each constraint $c \in S$, we associate a set σ_c of semi-valid inputs, defined as

$$\sigma_c = \{i \in \mathcal{I} \mid i \notin c \wedge \forall c' \in S \setminus \{c\}.\ i \in c'\} \ .$$

The set of semi-valid inputs is the union of all such σ_c:

$$I_{\mathsf{semi\text{-}valid}} = \bigcup_{c \in S} \sigma_c \ .$$

The set $\mathcal{I} \setminus (I_{\mathsf{valid}} \cup I_{\mathsf{semi\text{-}valid}})$ consists of what we call entirely-invalid inputs in § 1. Figure 1 shows an example where S consists of three constraints. The constraints collectively define the set of valid inputs, and associated to each constraint we define a set of semi-valid inputs.

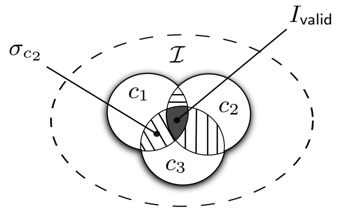

Figure 1: An example of valid and semi-valid inputs defined by $\{c_1, c_2, c_3\}$.

A constraint c in S is *redundant* iff $\bigcap_{c' \in S} c' = \bigcap_{c' \in S \setminus \{c\}} c'$. It is immediate that a constraint c is redundant iff c subsumes the intersection of the constraints in $S \setminus \{c\}$; that is $\bigcap_{c' \in S \setminus \{c\}} c' \subseteq c$. Note that for any redundant constraint c, we have $\sigma_c = \emptyset$ because if an input i violates c, then i must violate at least one additional constraint c', with $c' \neq c$. A set S of constraints is *minimal* if S does not contain redundant constraints; otherwise S is *non-minimal*. The set of valid inputs characterized by a non-minimal set S of constraints can be defined by a proper subset of S. We come back to the notion of non-minimality shortly.

A test set T is a finite set of tests. Each test defines an input value. We therefore identify each test set T with a finite subset of $\mathcal{I}$. Recall that we use the words *tests*, *test cases*, and *test inputs* interchangeably in the context of fuzz testing. Test t *covers* the set σ_c of semi-valid inputs iff $t \in \sigma_c$. For a test set T and a set S of constraints, we define

$$\mathsf{Cov}_S(T) = \{c \in S \mid \exists t \in T.\ t \in \sigma_c\} \ .$$

Note that $\mathsf{Cov}_S(T) \subseteq S$, for any T.

We are now ready to define our semi-valid input coverage criterion. Below, $|X|$ is the cardinality of the set X.

Definition 1. *The* semi-valid input coverage *criterion, denoted* SVCov*, is the function that maps any set S of constraints and any test set T to $\{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$, defined as:*

$$\mathrm{SVCov}_S(T) = \frac{|\mathsf{Cov}_S(T)|}{|S|} \ .$$

■

Note that SVCov conforms to the coverage criteria axioms given in § 2.1. In particular, for any S, we have $\mathrm{SVCov}_S(\emptyset) = 0$, and $\mathrm{SVCov}_S(T_1) \leq \mathrm{SVCov}_S(T_2)$ whenever $T_1 \subseteq T_2$. Also, for any minimal set S of constraints we have $\mathrm{SVCov}_S(T) = 1$ for $T = \cup_{c \in S} \sigma_c$. More importantly, for any S and $T \subseteq I_{\mathsf{valid}}$ we have $\mathrm{SVCov}_S(T) = 0$. Therefore SVCov indeed conforms to the fuzz-testing coverage axiom. In fact, valid inputs do not contribute to SVCov: for any finite $T \subseteq \mathcal{I}$ and finite $T' \subseteq I_{\mathsf{valid}}$, $\mathrm{SVCov}_S(T) = \mathrm{SVCov}_S(T \cup T')$.

A few remarks are due:

- The definition of SVCov is agnostic to the technique used to generate tests. The SVCov criterion can, for example, be used to measure the coverage of fuzz tests generated using mutation-based techniques, using grammars, and so forth; see [22, 32].

- The SVCov criterion depends only on the set S of constraints and the test set T. The SUT's structure, e.g. its source code, plays no role. In this sense, SVCov complements the existing criteria that refer to the SUT's code or model.

- A test set that is thorough with respect to SVCov may be inadequate. However we contend that a test set that is not thorough according to SVCov misses tests that can reveal faults in the SUT (see below). As discussed before, this observation is not specific to SVCov: it applies to all coverage criteria.

- Given a set S of constraints, maximum SVCov coverage is achievable, i.e. there is a test set T such that $\text{SVCov}_S(T) = 1$, iff S is minimal. This is because if S contains a redundant constraint c then $\sigma_c = \emptyset$.

- The amount of information provided by SVCov hinges upon the constraints in S. For instance, if S is a singleton, SVCov is rather uninformative because, for any T, the value of $\text{SVCov}_S(T)$ is either 0 or 1, excluding the values inbetween. In the context of our case study, we confirm that it is feasible and in fact straightforward to define a sufficiently detailed set of constraints for practically-relevant protocols.

We remark that the directed construction of test sets using the set of constraints to achieve high SVCov is possible. Depending on the nature of the constraints, test case generation may be automated, e.g., by negating each constraint, one at a time, and solving the resulting conjunction of the negated constraint with the remaining constraints. Directed test generation is however out of the scope of this paper, and therefore not further discussed here.

We claim that SVCov satisfies the three requirements given in § 1, namely *feasibility*, *relevance to coverage*, and *relevance to discovering faults*. To verify this claim, we have conducted a case study using the Internet Key Exchange protocol. In this case study, we have observed that investigating the causes for low SVCov can reveal problems in all components related to fuzz testing. By addressing these issues, we were able to extend our test set with tests that revealed a concrete vulnerability in the SUT. The case study is presented in § 3.

2.3 Guarded Constraints

We have observed that constraints defining actual systems often implicitly define *applicability* conditions, which the inputs must satisfy. For instance, the constraint "for any input, the third byte must be the xor of the first two bytes" is clearly applicable only to inputs of length greater than three bytes. We give examples of applicability conditions for IKE inputs in our study, presented in § 3. Below we introduce the notion of *guarded constraints* in order to account for applicability conditions.

A guarded constraint is a pair (c_g, c_t), denoted $c_g \triangleright c_t$, where c_g and c_t are constraints, i.e. $c_g \subseteq \mathcal{I}$ and $c_t \subseteq \mathcal{I}$. The subscripts g and t stand for *guard* and *target* respectively. Intuitively, c_g defines the inputs to which the guarded constraint $c_g \triangleright c_t$ is applicable. An input i *vacuously* satisfies $c_g \triangleright c_t$ iff $i \notin c_g$. An input i *non-vacuously* satisfies $c_g \triangleright c_t$ iff $i \in c_g \ \wedge \ i \in c_t$. An input i *violates* $c_g \triangleright c_t$ iff $i \in c_g \ \wedge \ i \notin c_t$; that is, i violates the guarded constraint iff the constraint is applicable to i and i violates c_t. A guarded constraint $c_g \triangleright c_t$ is applicable to all inputs in $\mathcal{I}$ iff $c_g = \mathcal{I}$.

Note that this definition of guarded constraints refines the definition of constraints given in § 2.2. That is, a guarded constraint $c_g \triangleright c_t$ specifies the constraint c defined by the set $(\mathcal{I} \setminus c_g) \cup c_t$. Therefore, the notion of SVCov and the definition of σ_c naturally extend to guarded constraints.

In § 3.4.3, we observe that valid inputs that vacuously satisfy a guarded constraint $c_g \triangleright c_t$ are not useful for generating semi-valid inputs in $\sigma_{c_g \triangleright c_t}$. That is, if the valid input $i \notin c_g$, then the mutated i is unlikely to belong to $\sigma_{c_g \triangleright c_t}$. Obviously, this observation is relevant only to mutation-based fuzz-testing techniques, such as the one described in our case study.

2.4 Measuring and Using SVCov

In this section we turn to the practical aspects of SVCov. We first describe how to measure SVCov and then we discuss the factors that typically affect SVCov.

The first step to measuring SVCov is to specify the set S of constraints that collectively define the SUT's valid inputs. Let T be the set of tests that are executed against the SUT. For each test in T, one checks how many constraints are violated by the test. If exactly one constraint is violated by a test, then the violated constraint is added to the set $\mathsf{Cov}_S(T)$. Finally, one calculates $\text{SVCov}_S(T)$ as in Definition 1. Algorithm 1 gives pseudo-code for this procedure.

Algorithm 1 Measuring SVCov

Input. A nonempty set S of constraints. A set T of tests.
Output. $\text{SVCov}_S(T)$.

```
$\mathsf{Cov} \leftarrow \{\}$
for all $t \in T$ do
  if $|\{c \in S \mid t \notin c\}| = 1$ then
    $\mathsf{Cov} \leftarrow \mathsf{Cov} \cup \{c \in S \mid t \notin c\}$
  end if
end for
return $|\mathsf{Cov}|/|S|$
```

Algorithm 1 requires that each constraint c in S is a recursive set: for each test t it must be decidable whether $t \in c$. Constraints are in practice often infinite sets (see § 3), and are therefore defined using their characteristic functions. That is, for each $c \in S$ there exists a computable function χ_c where $\chi_c(t) = 1$ if $t \in c$ and $\chi_c(t) = 0$ otherwise. The characteristic functions typically parse t and return 1 if t meets certain conditions. For instance, the characteristic function for the (artificial) constraint "for any input, the third byte must be the xor of the first two bytes" would parse the input, calculate the xor of the first two bytes and then compare the result with the third byte. If they are equal, then the function would return 1, and 0 otherwise. Procedural examples of characteristic functions are given in § 3.

After measuring SVCov for a set T of tests, with respect to a set S of constraints, the tester may decide that the coverage achieved by T is low. This decision in general depends on the risk analysis of the SUT, the time and resources available to the tester, etc. The tester must investigate the reasons for low SVCov to increase T's coverage. We have identified, through our case study, a number of factors that affect SVCov:

(1) *Implementation bugs in the fuzz-testing tool and imprecise fuzz operators.*

Suppose that to violate a given constraint, a certain bit in a field must be set to zero, while the fuzz operators can only replace all field's bits with zeros. In this case, the fuzz operators are imprecise for the task at hand if replacing the remaining bits with zeros leads to violating additional constraints. We give concrete examples of imprecision of fuzz operators in § 3.

To address this problem, the tester must fix the fuzz-testing tool and improve its fuzz operators.

(2) *Non-minimal set of constraints.*

It is evident that if a constraint c is redundant, then the constraint cannot be uniquely violated simply because $\sigma_c = \emptyset$; see § 2.2.

To address this problem, the tester must analyze the constraints in S and avoid any redundancies by rewriting or removing constraints.

(3) *Vacuous valid inputs.*

We have observed that the tests created by mutating valid inputs that vacuously satisfy the constraints in S are likely to result in a low SVCov. We come back to this observation in § 3.4.3.

To address this problem, the tester must modify the algorithm/program that generates the valid inputs. As mentioned before, this factor is relevant only to mutation-based fuzz-testing techniques.

(4) *Time.*

It is possible that fuzz testing can uniquely violate a constraint c, but the tests do not cover σ_c because fuzz testing is stopped prematurely.

To address this problem, the tester must take more time to fuzz-test the SUT.

How, and to what extent, the tester should address these issues depends on the available time and resources. The tester may decide to address (some of) the issues, and repeat fuzz testing the SUT.

SVCov versus the set of violated constraints. In order to have a reference point for measuring SVCov, we define a metric, dubbed Violated, that measures the total number of violated constraints. Formally, given a set S of constraints and a test set T, Violated is defined as

$$\text{VIOLATED}_S(T) = \frac{|\{c \in S \mid \exists t \in T.\ t \notin c\}|}{|S|} .$$

Note that Violated counts all the constraints that are violated by T, while SVCov refers to the number of *uniquely* violated constraints (see § 2.2). Therefore, if a test $t \in T$ violates two constraints in S simultaneously, then both constraints are counted in Violated, but neither is counted in SVCov. Obviously, for any S and T, $\text{VIOLATED}_S(T)$ is larger than $\text{SVCOV}_S(T)$.

The comparison of Violated to SVCov can be used as a guide for finding the reasons for low SVCov. The gap between Violated and SVCov suggests that either the fuzz operators are imprecise, i.e. they "degrade" the valid inputs to the extent that multiple constraints are violated, or that S is non-minimal; see factors (1) and (2) above. The constraints that are never violated suggest that either the valid inputs are vacuous (see § 2.3), or that some semi-valid inputs are never generated due to time outs; see factors (3) and (4) above.

The algorithm for computing Violated is similar to Algorithm 1 and hence omitted here.

3. SVCov FOR FUZZ TESTING IKE

The purpose of our case study is to investigate the feasibility and the benefits of measuring SVCov. Our SUT is OpenSwan [23], which is a mature open-source implementation of the Internet Key Exchange protocol (IKE). Implementations of security protocols, such as IKE, are known to be challenging to fuzz-test [32] because they have complex input structures, they use encryption, and they are stateful.

The rest of this section is organized as follows: in § 3.1 we describe our experimental setup, where we provide background on SecFuzz, the fuzz-testing tool used in the study. In § 3.2 we give the process and the heuristics we have used to extract constraints for IKE, along with a number of concrete examples. There we also discuss how SVCov accounts for IKE's stateful nature. We have used the constraints to measure SVCov of the tests generated by SecFuzz, and we summarize the results in § 3.3. We discuss the causes for low SVCov in § 3.4. In § 3.5 we present our improvements and report on a previously unknown vulnerability.

3.1 Experimental Setup

For our experiments we use SecFuzz [34], the tool we have previously developed for fuzz testing security protocol implementations. SecFuzz uses a concrete protocol implementation to generate valid inputs; we refer to this implementation as the *opposite endpoint.* SecFuzz mutates the valid inputs using a set of fuzz operators, and then the mutated inputs are sent to the SUT. The SUT's behaviors during testing are checked for failures.

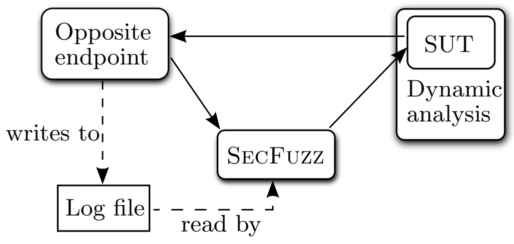

Figure 2: Experimental Setup. Messages sent to the SUT pass through SecFuzz. The opposite endpoint shares cryptographic information (e.g. keys and algorithms) with SecFuzz using a log file.

As shown in Figure 2, the SUT is one of the endpoints participating in the security protocol and the opposite endpoint is the other endpoint. For example, if the SUT is the protocol's initiator (often called Alice), then the opposite endpoint is the responder (often called Bob). Note that the two endpoints need not belong to the same software implementation of the protocol.

Table 1: SecFuzz's fuzz operators.

Category	Fuzz operator
Fuzz messages	Insert a well-formed message
Fuzz payloads	Insert a random payload
	Duplicate a randomly chosen payload
	Remove a randomly chosen payload
Fuzz fields	Set to a random number
	Set to zero
	Append random bytes
	Modify a random byte
	Set to the empty string
	Insert string termination

We configure the communication environment to route the messages destined for the SUT through SecFuzz. The opposite endpoint therefore generates and passes valid inputs to the fuzzer. The role of SecFuzz is to mutate the messages and forward them to the SUT. However, SecFuzz does not directly work with the messages it receives because they are normally encrypted. Indeed, mutating encrypted messages most certainly results in garbage, i.e. data the SUT would drop outright. We therefore have the opposite endpoint share its cryptographic information (e.g. keys, encryption and decryption libraries) with SecFuzz using a log file to allow SecFuzz to decrypt messages before mutating them. A detailed description of SecFuzz can be found in [34].

The SUT in our case study is the IKE responder implementation in OpenSwan v2.6.35 [23]. OpenSwan is a stable open-source Internet Protocol Security (IPsec) implementation, available for many major enterprise Linux distributions. OpenSwan is written in C and has 629,173 lines of code. The SUT is executed within a dynamic analysis tool, Memcheck [18], which serves as our test oracle. Memcheck is a memory error detector for C and C++ binary programs based on Valgrind [21]. It detects a wide range of memory errors such as use of undefined variables, invalid memory access, incorrect heap memory management, memory leaks, and others. Memory error detectors are widely adopted as test oracles for fuzz testing (see, e.g., [8, 12, 32]) because memory faults are security critical and often exploitable by attackers [33].

We used OpenSwan's initiator as the SUT's opposite endpoint for generating inputs. The inputs generated in a protocol run depend on the opposite endpoint's configuration files; these files refer to different protocol setups. To ensure that the opposite endpoint uses a range of different configurations, we let the opposite endpoint execute multiple times using different configuration files, which we automatically generate.

An input to a security protocol implementation, such as our test subject OpenSwan, is the sequence of messages exchanged during a protocol run. Each message typically consists of multiple payloads, where each payload consists of multiple fields. The protocol standard specifies how the fields' bits are interpreted. The layered structure of the inputs to protocol implementations allows SecFuzz to mutate the inputs at different levels of abstraction. SecFuzz's fuzz operators are described in Table 1. We remark that a message often has dependencies across its fields. For example, a string field with variable length may have a dedicated field indicating the string's length. Similarly, payloads often have a field indicating the type of the next payload in the message. The messages are often preprocessed by packet filters upon reception and messages with inconsistent fields are dropped. To ensure that not all mutated messages are dropped, SecFuzz may choose to update the dependent fields after applying a field/payload fuzz operator; see [34] for further details.

Table 2: Sample Constraints Extracted from IKE-related RFCs.

ID	Description
c_1	The first aggressive mode message must contain a key exchange payload.
c_2	If a message contains a proposal payload, then the proposal payload's next-payload field must be set to 2 or 0.
c_3	The length field correctly identifies the payload's length.

3.2 IKE Constraints

IKE is a widely deployed security protocol used to establish security associations between two endpoints. A security association (SA) is a set of cryptographic attributes (e.g. encryption algorithms and hash functions) and a security policy used to protect information. IKE uses the Internet Security Association and Key Management Protocol (ISAKMP), which is a framework for authentication and key exchange [17]. The ISAKMP specification defines the payload formats (e.g. proposal, key exchange, and identification payloads) used within IKE. IKE proceeds in two phases. The first phase has two different modes, *main* and *aggressive mode*, both of which set up a security association between the endpoints. The negotiated SA establishes a secure channel for further communication between the endpoints. The purpose of the second phase, called *quick mode*, is to set up an SA on behalf of another service, such as IPsec. After completing phase two, the two endpoints may exchange additional messages, e.g. to check whether the other endpoint is alive. These messages are specified as *information mode* exchanges. The IKE protocol has multiple versions specified in several RFC documents. For our case study we use IKEv1 specified in [13].

To measure SVCov for a set of IKE fuzz tests, we first must specify, using IKE's RFC documents, the set of constraints that define IKE's valid inputs. For this purpose, in addition to IKEv1 (RFC2409 [13]), we use IKE-relevant subsets of the ISAKMP protocol (RFC2408 [17]) and also the Internet IP Security Domain of Interpretation for ISAKMP (RFC2407 [25]). RFC documents are informal and often lengthy, for example RFC2408 is 86 pages. We use the standard keywords *must*, *must not*, and *required*, as heuristics to analyze IKE's RFCs. These keywords not only point at the important sentences in the specifications, but also they most often have an unambiguous interpretation, which makes them good candidates for specifying constraints. Examples of IKE constraints are given in Table 2. Constraint c_2, for instance, is quoted from RFC2408.

Using the straightforward process described above, we extracted 217 IKE constraints from the aforementioned RFCs.

```
class C002(object):
   ...
   def check(self, msg, prev_msgs):
      if msg[Payload.Proposal] is None:
         return VACUOUS
      if msg[Payload.Proposal].next_payload not in [0,2]:
         return FALSE
      return TRUE
```

Figure 3: Encoding of constraint c_2, from Table 2, in Python.

The number of constraints per main, aggressive, quick, and information mode are 82, 68, 55, and 12, respectively. The time required to extract the constraints was approximately 8 person-hours. The constraints are publicly available.[1]

Next, the informal constraints we have extracted from IKE-related RFCs are modeled as guarded constraints. For instance, constraint c_2 (in Table 2) is applicable only to the inputs that contain a proposal payload. We represent c_2 as $c_g \triangleright c_t$, where c_g consists of all the inputs that contain a proposal payload and c_t consists of all the inputs whose proposal payload's next-payload field is set to 2 or 0. Guards in the IKE constraints typically check whether the constraint is applicable to a particular IKE exchange mode (quick, main, or aggressive mode), or to a specific payload (e.g. SA payload, proposal payload, key exchange payload, etc.), or to a specific message ordering (e.g. c_1 in Table 2).

Note that to fuzz-test stateful SUTs thoroughly, one needs to generate semi-valid inputs that explore the system in depth. In the case of IKE, for instance, to fuzz-test the implementation of the protocol's second phase (namely, its quick mode), the first phase of the protocol must be successfully completed. Indeed, the guarded constraints that refer to quick mode are applicable only to the inputs that successfully complete IKE's first phase. In order to violate such a constraint, the test would therefore need to complete IKE's first phase, and then violate the constraint's target. The notion of SVCov thus accounts for states in IKE: the tests that do not explore the SUT in depth would not violate the constraints that refer to quick mode (or any following mode, for this reason).

To check the constraints automatically, we have encoded them using Scapy, a Python library for parsing and manipulating messages [28]. Figure 3 shows a sample implementation. The method *check* in Figure 3 accepts two arguments: a message and the sequence of messages preceding the message. The second argument is needed because some constraints refer to the previously exchanged messages (during the same protocol run). We have implemented a constraint checker that takes an IKE test, i.e. a sequence of IKE messages, and outputs the sets of violated, vacuously satisfied, and non-vacuously satisfied IKE constraints.

We measured the time required to check the constraints on a machine with an i7-2600 quad-core processor and 8GB of RAM. On average, the constraint verifier takes 41 milliseconds to check all 217 constraints for one IKE input, which is 0.19 milliseconds per constraint. In comparison, it takes on average 1 second to execute a single test (while running Memcheck). The overhead of measuring the coverage is therefore negligible.

[1]http://www.infsec.ethz.ch/research/software/secfuzz

We conclude that, in the context of IKE, SVCov meets requirement R1 given in § 1: it is feasible, in a reasonable amount of time, to precisely define the set of semi-valid inputs for any IKE implementation. Moreover, the overhead for checking satisfiability of the constraints is negligible. Note that extracting the constraints that define the set of valid inputs of IKE is a one-off task: the constraints are neither bound to any particular implementation of IKE, nor to any specific technique used to (fuzz-)test IKE.

3.3 SVCov Measurements

We used the constraints defined in § 3.2, hereafter referred to as S_{IKE}, to measure the semi-valid input coverage of 36,000 tests generated using SecFuzz. The measurements presented below correspond to running SecFuzz for 10 hours. That is, executing each test requires on average 1 second.

In the graph depicted in Figure 4, the solid line shows $\text{SVCov}_{S_{\mathsf{IKE}}}(T)$, and the dashed line shows $\text{Violated}_{S_{\mathsf{IKE}}}(T)$, both as a function of the number of tests executed on the SUT. The graph shows that initially the tests generated by SecFuzz rapidly increase SVCov. As expected, the rate of increase decreases as fuzz testing progresses. This is because initially the sets σ_c are not covered and the tests are very likely to cover them. As more σ_c are covered by the tests, the newly generated tests are increasingly likely to fall into already covered σ_c and hence they do not increase SVCov.

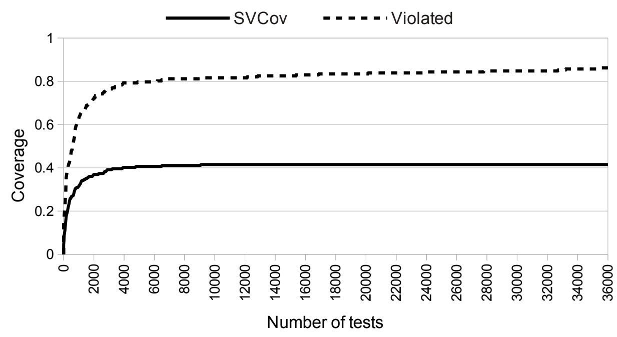

Figure 4: SVCov versus Violated.

At the end of the experiment, SVCov is 90 out of the 217 constraints, that is 41%. Moreover 86% of IKE's constraints are violated at least once and therefore 14% of the constraints in S_{IKE} are never violated. In § 3.4, we explain why SVCov is low and why there is a large discrepancy between SVCov and Violated.

We have discovered a critical security vulnerability in the SUT (OpenSwan's responder) during this experiment, which we reported in CVE-2011-4073. The vulnerability is found with 4,800 tests, on average. We remark that fuzz testing is inherently random. Therefore, to measure the average time SecFuzz requires to find the aforementioned vulnerability, we have executed the tests several times.

3.4 Coverage Analysis

In this section, we analyze the coverage achieved in our experiment, reported in § 3.3. The purpose of the analysis is to understand why SVCov is low and to find ways to increase it.

The definition of SVCov depends on the set S_{IKE} of constraints and on the test set. In turn, the test set generated by SecFuzz depends on the set of valid inputs provided by the opposite endpoint and the set of fuzz operators implemented in SecFuzz. Therefore, three factors affect SVCov: (1) the set of fuzz operators implemented in SecFuzz, (2) the set of constraints S_{IKE}, and (3) the set of valid inputs. Below, we discuss the issues related to each of these factors that we have found. Naturally, the time we allocated for fuzz testing IKE also plays a role.

3.4.1 Implementation Bugs in SecFuzz and Imprecise Fuzz Operators

As mentioned above, there is a large gap between SVCov and Violated in the experiment. The fuzz operators of SecFuzz could be the reason: the operators "degrade" the valid inputs to the extent that the mutated inputs cannot uniquely violate certain constraints in S_{IKE}. Guided by the constraints that were violated, but not uniquely, in the experiment, we have manually investigated the fuzz operators of SecFuzz, and discovered several problems:

Scapy Library: A given IKE payload specifies encryption attributes, which SecFuzz cannot modify as it relies on Scapy. Consequently, if a constraint requires that this payload specifies a particular encryption attribute, then SecFuzz cannot generate tests that uniquely violate the constraint. This is a limitation SecFuzz inherits from the Scapy library.

Insert Payload Operator: Due to its randomness, SecFuzz's insert payload fuzz operator does not always insert payloads that are well-formed according to IKE. For instance, in order to violate the IKE constraint "*a message that contains a security association payload must not contain any identification payloads*," SecFuzz's insert payload operator must be used. However, the inserted identification payload is with an overwhelming probability ill-formed. Therefore, the constraint is unlikely to be uniquely violated by the tests generated using SecFuzz.

Implementation Bugs: We have identified a number of implementation bugs in SecFuzz. For instance, for some operators, the random numbers that can be placed in a field are limited to positive integers smaller than 100, while to violate some IKE constraints SecFuzz must insert numbers greater than 100.

Overall, using SVCov as a guide, we pinpointed a number of subtle problems in SecFuzz's fuzz operators. In principle, some of these problems could have been found by testing the fuzz operators (and, the fuzz operator tester needs to be tested too). However, testing the operators is limited to revealing bugs in their implementation and cannot discover missing fuzz operators or issues with their precision.

3.4.2 Non-minimal Set of Constraints

For any constraint c, σ_c obviously cannot be covered if it is empty; see § 2.2. By analyzing the constraints in S_{IKE}, we have found one constraint c where $\sigma_c = \emptyset$. The constraint c states that: *The ID field in the first round messages must be set to zero.* This constraint subsumes the intersection of the following two constraints:

- c_p: *The first round messages must contain the key exchange payload.*
- c_q: *The ID field must be set to zero in messages containing the key exchange payload.*

If c is violated, then the message is a first round message and the ID field is not set to zero. However, if the message does not contain a key exchange payload, it violates c_p; if it does contain a key exchange payload, then it violates c_q.

We did not remove c from S_{IKE}, so that a fair comparison can be made between the first experiment and our second experiment, discussed in § 3.5. Note that the overlapping constraints discussed above are not a problem of S_{IKE} per se: they indicate that the corresponding phrases in the RFCs are redundant.

3.4.3 Vacuous Valid Inputs

We noticed that the valid inputs that vacuously satisfy a guarded constraint $c_g \triangleright c_t$ are ill-suited for generating semi-valid inputs that belong to $\sigma_{c_g \triangleright c_t}$. Suppose that the constraint is vacuously satisfied by an input i, i.e. $i \notin c_g$. Let us write $\hat{\imath}$ for the mutated input that SecFuzz sends to the SUT. In order for $\hat{\imath}$ to belong to $\sigma_{c_g \triangleright c_t}$, the SecFuzz's fuzz operators should "enhance" i to satisfy c_g and simultaneously they should "degrade" i to violate c_t. This is clearly beyond the purpose of the fuzz operators. Indeed, in our case study, 79% of the constraints $\sigma_{c_g \triangleright c_t}$ are always uniquely violated by tests generated using valid inputs that satisfy c_g.

As an indicator for the valid inputs' quality we can therefore measure the number of constraints that are vacuously satisfied by all the valid inputs. Note that we measure the vacuity of the *valid* inputs, i.e. the ones created by the opposite endpoint; we do not measure the vacuity of the tests, i.e. the mutated inputs SecFuzz sends to the SUT. We found that 22 constraints (10%) are vacuously satisfied by all the valid inputs. This indicates that there are missing valid inputs in our experiment.

Missing Valid Inputs. 18 of the 22 constraints are vacuously satisfied because the inputs generated by the opposite endpoint have a fixed ordering of the payloads in the message. For instance, the key exchange payload, which carries the data to generate a session key, is never placed as the last payload in the message. According to IKE's specification, there are however no ordering restrictions for the payloads in IKE messages (with a few exceptions for the messages exchanged in quick mode). However, the opposite endpoint's implementation (OpenSwan's initiator) orders the payloads deterministically, which causes 18 constraints to be always vacuously satisfied.

The other 4 constraints are vacuously satisfied because the endpoints always use their IPv4 addresses for identification. Consequently, constraints similar to: *An identification payload of type IPv6 must specify a valid IPv6 address*, are always vacuously satisfied. Recall that the generated valid inputs depend on the configuration files used to initialize the opposite endpoint (see § 3.1). We noted that in our experiment there were no configuration files that specify identification types different from IPv4. For example, there were no configuration files for IPv6 and ASN.1 X.500 Distinguished Name.

To generate valid inputs that do not vacuously satisfy the constraints in S_{IKE}, we have implemented a message preprocessor that shuffles the payloads in the message before handing them to SecFuzz, and we have created additional configuration files for the opposite endpoint to generate valid inputs using different identification types. These modifications are indeed sufficient: no constraints are vacuously satisfied by the valid inputs, after the improvements.

3.4.4 *Coverage Analysis Summary*

We have carefully analyzed the results of the experiment. The analysis, guided by SVCov, led to a number of significant improvements in SecFuzz, its fuzz operators, and its pre-processing libraries. Moreover, it helped us to discover a redundancy in the RFCs themselves. Indeed, SVCov provided us with the means to pinpoint the aforementioned problems, and guided us in their repair. We conclude this discussion by pointing out that requirement R2, given in § 1, is met in the context of fuzz testing IKE. In the next section, we see how these improvements translate to more effective fuzz testing of IKE with higher SVCov.

3.5 Measuring SVCov After Improvements

We have repeated the experiment of fuzz testing IKE after addressing the implementation bugs in SecFuzz (with the exception of the insert payload fuzz operator in SecFuzz because its repair is nontrivial) and the problem of missing valid inputs. We then measured SVCov as before, depicted in Figure 5. The new measurements show that SVCov is increased from 41% to 89%. That is, 193 out of the 217 constraints have been uniquely violated. Note that the increase in SVCov is not due to enlarging the test set: in both experiments we have 36,000 tests.

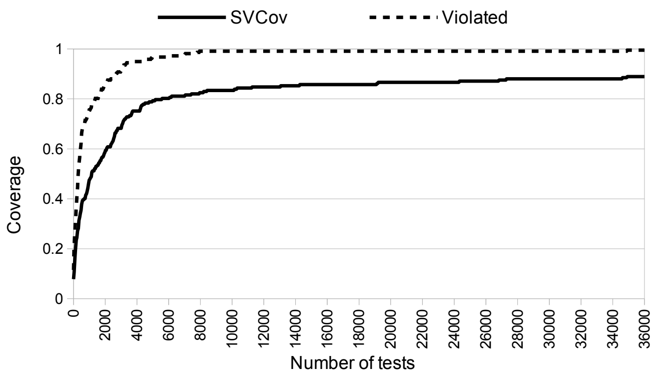

Figure 5: SVCov versus Violated: Second Experiment.

In terms of Violated, 216 of the 217 constraints are violated by the tests. Therefore, 23 constraints are violated, but not uniquely. For 11 of these constraints, the reason for always violating them along with additional constraints is the limited precision of the insert payload fuzz operator. One constraint cannot be uniquely violated because of a redundancy in S_{IKE}; see § 3.4.2. The corresponding semi-valid inputs for the remaining constraints were not generated due to timeout: we stopped the experiment after 10 hours.

A Vulnerability. The purpose of fuzz testing is to find faults in the SUT, rather than increasing SVCov per se. In the experiment conducted after improving SecFuzz, we discovered a previously unknown security vulnerability in OpenSwan's responder. The vulnerability was found with 34,000 tests on average, and it concerns an invalid read in the responder when an aggressive mode message is received with an ID payload that refers to the ASN.1 X.500 Distinguished Name. When the field storing the identity is omitted, OpenSwan accesses unallocated memory. Unallocated memory accesses are dangerous as they can, with a high likelihood, be exploited by attackers [33]. We have communicated the details of this vulnerability to the OpenSwan development team.

The test exposing the problem is generated by the *set to the empty field* fuzz operator. The valid input required to generate the test case is an aggressive mode message with an ID payload using distinguished names. The reason the vulnerability was not discovered in our previous experiment is that the valid input needed to create the test was not generated by the opposite endpoint; see § 3.4.3.

The vulnerability is revealed by a test input that uniquely violates a constraint in S_{IKE}. This provides evidence supporting requirement R3: increasing fuzz-testing coverage in terms of SVCov exposes more faults in the SUT.

3.6 Threats to Validity

In our case study, we used a security protocol implementation as our test subject. We can therefore draw only limited conclusions on the feasibility of defining a rich set of constraints for other kinds of test subjects. We believe however that it is feasible to define constraints for any software system for which there exists a sufficiently precise or formal specification. This is the case for any software implementing standards (for example PDF viewers and HTTP services), software developed following design-by-contract methodology (APIs), software developed using a model-driven approach, and so forth.

We have used a mutation-based fuzz-testing tool, SecFuzz, for fuzz testing our test subject. As noted in § 2.2, SVCov is agnostic to the technique used to generate tests and therefore the usefulness of SVCov extends to other fuzz-testing techniques. The test oracle used in our case study is Valgrind's Memcheck. Memory error detectors such as Memcheck are widely used in fuzz testing; cf. [8, 12]. This observation mitigates the threats to the external validity of our study.

4. RELATED WORK

We are not aware of any existing coverage criterion specific to fuzz testing. Below, we compare SVCov to the coverage criteria that have been applied to fuzz testing. We are in agreement with Myers et al. that to generate robust test sets one must use multiple coverage criteria to balance their individual weaknesses [20]. That is, we see SVCov as a complementary coverage criterion to the existing ones.

Program-based coverage criteria such as statement, decision, and condition coverage (e.g. see [7, 20, 36]) have been used to measure the thoroughness of fuzz tests. Examples include [4, 10, 31]. In contrast to these metrics, SVCov is agnostic to the program structure; it measures to what extent tests cover the domain of semi-valid inputs.

Input-based coverage criteria defined using input domain partitions and their boundary values have also been studied in the literature; see, e.g., [20, 36]. These criteria are similar to SVCov as they also measure coverage based on the inputs as opposed to the SUT's internal structure. However, the existing input-based criteria do not account for semi-valid inputs. For example, Kaksonen and Takanen discuss a coverage metric which reflects what portion of the input structure has been mutated by fuzz tests [15]. This metric is input-based, but does not account for semi-valid inputs. Similarly, Alrahem and Harris partition the input domain based on the SUT's specification, and then select fuzz tests from each partition [2]. The difference between valid, semi-valid, and entirely-invalid inputs is however not explicit in their partitions. As another example, Zhu et al. present data mutation operators for mutating CAMLE model diagrams [29]. They define validity constraints for the inputs and measure how many tests violate the constraints. In contrast to SVCov, they do not distinguish between the tests that violate a unique constraint, and those that violate more than one constraints. The distinction is essential for fuzz testing.

Takanen et al. [32] discuss quality assurance metrics for comparing fuzzers based on (1) their ability to detect known vulnerabilities, (2) the number of the supported protocols, and (3) the expected defect count, estimated using a fuzzer's historical fault detection success. These metrics are orthogonal to SVCov. The authors also suggest using protocol RFCs to build "negative" fuzz tests. We do not use RFCs for generating tests; rather, SVCov measures the coverage of such negative fuzz tests.

Opstad et al. present a comprehensive empirical study on fuzz testing more than 250 parsers using different tools [24]. Their goal is to find heuristics for deciding when to stop fuzz testing. They do not define coverage criteria.

Advanced fuzz-testing techniques such as white-box fuzz testing (e.g. [6, 9, 11, 12]), model-based fuzz testing (e.g. [1, 3, 16]) and mutation-based fuzz testing of cryptographic protocols [34], are different methods for fuzz testing a software system. SVCov defines how one can measure the coverage of test sets generated using such fuzz-testing methods.

5. CONCLUSIONS AND FUTURE WORK

We propose the semi-valid input coverage criterion SVCov for measuring the coverage of fuzz testing. In a case study on fuzz testing the Internet Key Exchange protocol (IKE), we show that SVCov is straightforward to measure and the coverage measurement overhead is negligible. Our case study also demonstrates that SVCov is informative and can guide the tester to improve fuzz-testing coverage. Moreover, improvements we have made to fuzz-testing tools and libraries to increase SVCov, led to discovering a previously unknown vulnerability in a popular, stable implementation of IKE.

Overall, our initial experiences with SVCov are very encouraging. Next, we plan to extend our experiments to a larger set of security-related protocols, in order to be able to draw statistically significant conclusions about the value of SVCov.

In terms of generalizing SVCov, one can extend the definition of semi-valid inputs to those that violate up to k out of n constraints. In this paper, we have focused on $k = 1$. Similarly, thorough test sets could be defined as those containing at least j semi-valid inputs per σ_c, for every constraint c. We have focused on $j = 1$ in this paper. We intend to investigate the cost-benefit ratio of such extensions in the context of security protocols.

Although the definition of SVCov is agnostic to the techniques used to generate tests, it would be interesting to apply SVCov to different fuzz-testing techniques; for example, to measure the coverage of tests generated by white-box fuzzers, or model-based fuzzers. This is an interesting topic for future research because we expect that the activities required to increase the SVCov of tests generated by, say, a white-box fuzzer, is different from the activities described in this paper.

6. ACKNOWLEDGMENTS

This work has been partially supported by the EU FP7 project SPaCIoS (no. 257876). We thank Marco Guarnieri, Srdjan Marinovic, and the anonymous reviewers for their helpful comments on the paper.

7. REFERENCES

[1] T. Alrahem, A. Chen, N. DiGiuseppe, J. Gee, S.-P. Hsiao, S. Mattox, T. Park, and I. Harris. INTERSTATE: A Stateful Protocol Fuzzer for SIP. In *Defcon 15*, pages 1–5, August 2007.

[2] T. Alrahem and I. Harris. Achieving Domain Coverage with Directed Fuzz Testing. Technical report, University of California, Irvine, CA, USA, June 2007.

[3] G. Banks, M. Cova, V. Felmetsger, K. C. Almeroth, R. A. Kemmerer, and G. Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *ISC*, pages 343–358, 2006.

[4] S. Becker, H. Abdelnur, J. L. Obes, R. State, and O. Festor. Improving Fuzz Testing Using Game Theory. In *Proceedings of the 2010 Fourth International Conference on Network and System Security*, NSS '10, pages 263–268, Washington, DC, USA, 2010. IEEE Computer Society.

[5] E. Bouwers, J. Visser, and A. van Deursen. Getting What You Measure. *Commun. ACM*, 55(7):54–59, July 2012.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[7] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Trans. Softw. Eng.*, 15(11):1318–1332, Nov. 1989.

[8] W. Drewry and T. Ormandy. Flayer: Exposing Application Internals. In *Proceedings of the first USENIX workshop on Offensive Technologies*, WOOT '07, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[9] V. Ganesh, T. Leek, and M. Rinard. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[10] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[12] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[13] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), Nov. 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[14] IEEE Standards Board. IEEE standard glossary of software engineering terminology, September 1990. Std 610.12-1990.

[15] R. Kaksonen. Test Coverage in Model-based Fuzz Testing, 2012. Presented at Model-based User Conference, Tallinn, Estonia.

[16] R. Kaksonen, M. Laakso, and A. Takanen. System Security Assessment through Specification Mutations and Fault Injection. In *Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues of the New Century*, pages 27–, Deventer, The Netherlands, 2001. Kluwer, B.V.

[17] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). RFC 2408 (Proposed Standard), Nov. 1998. Obsoleted by RFC 4306.

[18] Memcheck: A Memory Error Detector. http://valgrind.org.

[19] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33:32–44, December 1990.

[20] G. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. ITPro collection. Wiley, 2011.

[21] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[22] P. Oehlert. Violating Assumptions with Fuzzing. *IEEE Security and Privacy*, 3(2):58–62, Mar. 2005.

[23] OpenSwan. https://www.openswan.org.

[24] L. Opstad, J. Shirk, and D. Weinstein. Fuzzed Enough? When It's OK to Put the Shears Down, 2008. Presented at BlueHat v8, Redmond, WA, USA.

[25] D. Piper. The Internet IP Security Domain of Interpretation for ISAKMP. RFC 2407 (Proposed Standard), Nov. 1998. Obsoleted by RFC 4306.

[26] RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification, December 1992.

[27] J. Rushby. Automated Test Generation and Verified Software. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, pages 161–172. Springer-Verlag, Berlin, Heidelberg, 2008.

[28] Scapy. http://www.secdev.org/projects/scapy/.

[29] L. Shan and H. Zhu. Generating Structurally Complex Test Cases By Data Mutation: A Case Study Of Testing An Automated Modelling Tool. *Comput. J.*, 52(5):571–588, 2009.

[30] M. Staats, G. Gay, M. Whalen, and M. Heimdahl. On the Danger of Coverage Directed Test Case Generation. In *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 409–424, Berlin, Heidelberg, 2012. Springer-Verlag.

[31] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 1 edition, July 2007.

[32] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[33] The MITRE Corporation. CWE-List (2.1), Sept. 2011.

[34] P. Tsankov, M. Torabi Dashti, and D. Basin. SecFuzz: Fuzz-testing Security Protocols. In *7th International Workshop on Automation of Software Test (AST '12)*, pages 1–7. IEEE, June 2012.

[35] E. J. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Trans. Softw. Eng.*, 12(12):1128–1138, Dec. 1986.

[36] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997.