Hermes: A Targeted Fuzz Testing Framework

Caleb Shortt(✉) and Jens Weber

University of Victoria, Victoria, Canada
{cshortt,jens}@uvic.ca

Abstract. Security assurance cases (security cases) are used to represent claims for evidence-based assurance of security properties in software. A security case uses evidence to argue that a particular claim is true, e.g., buffer overflows cannot happen. Evidence may be generated with a variety of methods. Random negative testing (fuzz testing) has become a popular method for creating evidence for the security of software. However, traditional fuzz testing is undirected and provides only weak evidence for specific assurance concerns, unless significant resources are allocated for extensive testing. This paper presents a method to apply fuzz testing in a targeted way to more economically support the creation of evidence for specific security assurance cases. Our experiments produced results with target code coverage comparable to an exhaustive fuzz test run while significantly reducing the test execution time when compared to exhaustive methods. These results provide specific evidence for security cases and provide improved assurance.

Keywords: Security · Assurance · Fuzzing · Testing · Genetic-algorithms · Evidence

1 Introduction

Assurance is confidence that an entity meets its requirements based on evidence provided by the application of assurance techniques [2]. *Security assurance* narrows this scope to focus on security claims or requirements. Security assurance cases (security cases) are used to argue the assurance of specific claims. They provide a series of evidence-argument-claim structures that, when combined, provide assurance on the original claim.

A popular tool in security assurance is fuzz testing [6,8–10]. Fuzz testing is an automated type of random, or semi-random, negative testing that attempts to cause a target system to crash, hang, or otherwise fail in an unexpected manner [4–7]. It takes a dynamic analysis approach and tracks the attempted input and the resulting response from the system – whether or not it fails and, in some cases, includes the type of failure. In essence, it is a black-box "scattergun" approach where the accuracy of the "gun" is determined by the fuzzer utilized.

Due to the undirected nature of traditional fuzz testing methods, the evidence provided by fuzz tests is not specific to particular types of defects and thus fuzz testing results are only weakly linked to specific security claims. Traditional fuzz testing tools encounter difficulties, and become ineffective, if most generated

H. Fujita and G. Guizzi (Eds.): SoMeT 2015, CCIS 532, pp. 453–468, 2015.
DOI: 10.1007/978-3-319-22689-7_35

inputs are rejected early in the execution of the target program [12]. Random testing usually provides low code coverage [13]. While it is possible to afford a large amount of resources (time and/or computational) increase the coverage of fuzz tests, it would be more economically feasible to be able to target the fuzzer to particular security concerns in order to provide stronger evidence for specific assurance cases.

Directed fuzz testing approaches exist and include solutions that rely on taint analysis, symbolic execution, and constraint-solvers to provide a certain level of introspection [12,14,15]. These approaches are certainly an improvement over undirected fuzz testing in the quality of evidence provided, but the issues of performance, complexity, and uncertainty of application in "real world" systems leave much to be desired. It is an open question if symbolic execution fuzz testing can consistently achieve high code coverage on "real world" applications [14], and the symbolic execution "is limited in practice by the imprecision of static analysis and theorem provers" [13]. Finally, fuzz testing is executed for a certain amount of time to be considered "good enough". However "good enough" is a subjective term and lacks the quantitative properties required to be reviewed as evidence.

In this paper, we present a method for targeted fuzz testing that combines the input of static code analysis with an optimization function (based on Genetic Algorithms) that utilizes dynamic code coverage analysis. Our method has been implemented in a tool prototype (called Hermes) and evaluated in a case study using a real-world software system (Crawler4j) [31].

The purpose of our research is to investigate the questions: "Is it possible to use targeted fuzz testing to provide targeted evidence for security assurance cases? Is it also possible to reduce the computation time required while achieving the same code coverage as a full fuzz test run?".

Our evaluative analysis of software (the Crawler4j Java library [31]) using Hermes produced promising results and achieved near-parity code coverage when compared to an exhaustive, undirected, fuzz test – or full fuzz test, but each evaluation was able to do so in reduced execution time.

There rest of this paper is structured as follows. The next section discusses background and related work. We introduce Hermes in Sect. 3 and discuss our evaluation method in Sect. 4. Section 5 present the results of our evaluation experiment. Finally, we close with concluding remarks and pointers to future work in Sect. 7.

2 Related Work

2.1 Security Assurance Cases

"Assurance is confidence that an entity meets its requirements based on evidence provided by the application of assurance techniques" [2]. *Security assurance* narrows this scope to focus on security claims or requirements. Security assurance cases (security cases) are used to argue the assurance of specific claims. They provide a series of evidence-argument-claim structures that, when combined, provide assurance on the original claim.

For example, the claim "the REST API is secure against attack" may not be provable directly. Therefore the main claim will have to be decomposed into a set of *subclaims* that must be assured to assure the main claim. Once the subclaims are assured, the main claim can be considered assured. These subclaims would include statements such as "The use of the REST API cannot cause a buffer overflow". This subclaim would require either more subclaims that must be assured, or it must provide evidence that sufficiently support the claim.

The types of evidence vary widely depending on design and environment of a system, however some common types of evidence for security cases include black-box testing results, white-box testing results, model checking, standards compliance check lists, fuzz test results, and penetration-test reports.

2.2 Fuzz Testing

Fuzz testing is an automated type of random, or semi-random, negative testing that attempts to cause a target system to crash, hang, or otherwise fail in an unexpected manner [4–7]. It takes a dynamic analysis approach and tracks the attempted input and the resulting response from the system – whether or not it fails and, in some cases, includes the type of failure. In its traditional form, it is a black-box scattergun approach where the accuracy of the "scattergun" is determined by the fuzzer utilized.

Fuzz testing has proved to be a valuable addition to current software security techniques and has caught the attention of industry leaders such as Microsoft who have incorporated it into the Security Development Lifecycle (SDL) [9,10]. It is particularly well-suited to discover finite-state machine edge cases via semi-malformed inputs [6,8]. The partially-correct inputs are able to penetrate the initial layers of verification in a system and test the bounds of areas that may have not been considered by the developers or design team. These partially-correct inputs can be generated from inputs provided to the fuzzer at runtime where it uses it as a template, or they can be "mutated" from capturing input information that is known to be correct. These two methods define the two categories of fuzzers: "Mutation-based" and "generation-based" [6,8].

Generation-based fuzzers use random or brute-force input creation and are usually customized to generate variations of a particular protocol model or application data format that an application uses. Once the fuzzer is connected to the target it can generate its inputs and track the responses returned [8].

Mutation-based fuzzers do not incorporate a model for generating inputs but rather use random mutations on a library of known valid inputs [32]. The mutation process may include checksum calculation and other more advanced methods to penetrate the application's primary level of input validation [32]. Mutation-based fuzzers are considered "generic fuzzers" as the need to customize them to a particular target application is minimal [29]. Limited customization is possible to target specific parts of the input data format (or protocol) by using a "block-based approach" that segments the input into separate blocks [33]. Each block can either be fuzzed or left in its original state. With a block-based approach,

additional information blocks can be created and reused to construct various protocol definitions, file formats, or validation techniques such as checksums [34].

Directed Fuzzers and Optimization. Directed fuzz testing utilizes methods to optimize the mutation or generation of inputs, and include solutions that rely on taint analysis to provide a certain level of introspection [12,15]. Taint analysis relies on a "clean" run to provide a baseline execution pattern for the target application. It then compares all subsequent executions to the baseline in an attempt to find discrepancies. Further approaches include the addition of symbolic execution and constraint solvers to take full advantage of the introspective properties of the taint analysis approach [14].

Various techniques can be used for optimizing directed fuzzers with respect to a defined utility function, such as code coverage. Genetic Algorithms (GA) are one particularly well suited optimization technique in this context as the concept of a "genetic information string" defining a vector of features that are switched on and off during the generation or mutation of fuzzed input provides a natural fit. The idea behind GA's is modelled after the natural evolutionary process [17,18]. A GA is used to simulate the evolutionary progress of a population towards a certain "fitness" goal [19,36]. A "population", in this case, can be any group of features (called a feature string, individual, genotype, or chromosome) that are evaluated to provide a "fitness value" [20]. An evaluation function must be defined which provides one, or many, performance measures for a given feature string. The fitness function then determines which feature strings are most "fit" and should be used for creating the next-generation population [19,20]. Once a subset of feature strings is selected the next generation is created by mutation and crossover (also called "mating") [17,40,41]. An important advantage of a GA is that it is able to manipulate numerous strings simultaneously. This greatly reduces the chances of the optimization becoming stuck in a local minima [17].

3 The Hermes Approach

3.1 Overview

The approach we have implemented within Hermes falls under the category of directed, generation-based fuzzing and combines static and dynamic code analysis along with an optimization process that utilizes Genetic Algorithms. We will first provide an overview on our method and then discuss its elements in more detail. It is assumed that our fuzz testing method is used to generate evidence for security assurance cases, e.g., the ability of a software component to resist exploitation of certain types of security vulnerabilities. Once the vulnerabilities to target have been defined, our method starts with applying a static code analyzer to identify source code locations that may be vulnerable to specific types of security threats. The goal of our smart fuzzer is then to optimize test case generation to maximize coverage of the code that contains the potential vulnerabilities. Since our fuzzer is generation-based, it has a model of the input data (or protocol) of the software to be tested. Specific features in this model may

be switched on or off depending on a binary string that controls the random test case generation process. A first set of such feature strings are randomly generated and referred to as the *first generation* of tests. Consequently, the fuzz testing framework runs the target software with the test cases generated for the first generation of tests. The code coverage of the vulnerable target areas (determined by the static analyzer) is analyzed for these initial test runs. The feature strings that yield the highest code coverage are selected and used for producing of the next generation of feature strings, using a GA.

3.2 Design

The design of Hermes is based on a client-server architecture which facilitates both remote and local testing of targets, cf. Fig. 1. The processing for both the genetic algorithm and the fuzz test generation (using Sulley [43]) happens on the server side while the client is a thin wrapper that includes the code coverage tool (EMMA [44]). The client monitors the target and sends the coverage metrics to the server to complete the asynchronous loop of test, measure, and revise.

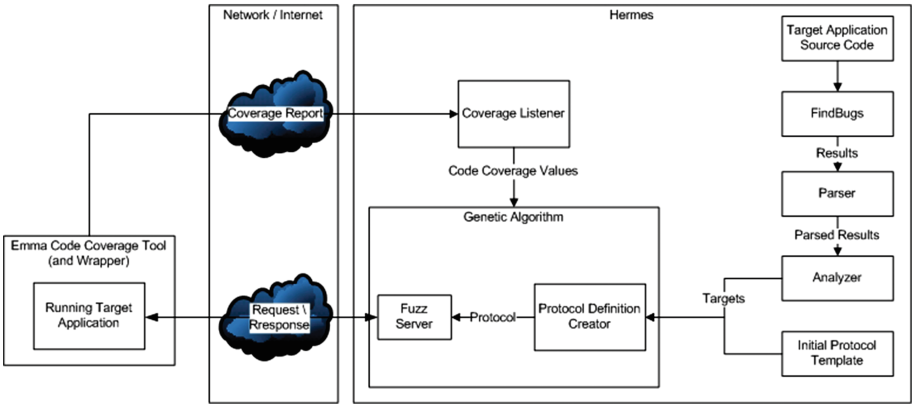

Fig. 1. A view of Hermes' architecture

Hermes relies on static analysis to find the initial sections of code with potential defects. It then parses, sorts, and identifies the target code that the framework will focus on. The list of targets and a protocol definition that specifies how to dynamically generate a language-specific protocol are passed to the genetic algorithm to begin the analysis. Hermes is able to dynamically generate a protocol for a specific language, with certain features included or excluded, start or stop the fuzz server, and revise the protocol based on code coverage feedback and the genetic algorithm's suggestions.

Protocol generation is achieved by using a protocol template that specifies a language's syntax in the Sulley protocol language [43]. Each feature is defined

using the language's syntax as building blocks. This allows Hermes to select which features to include or exclude to create a new protocol. A feature string determines the inclusion or exclusion of a protocol. The feature strings specified in Hermes include seven unique features for the target language – in our evaluation, HTML. The HTML features are anchor tags, image tags, div tags, iframe tags, object tags, javascript tags, and applet tags. Thus a feature string with all features included would look like the following: (1, 1, 1, 1, 1, 1, 1).

4 Evaluation

Our evaluation procedure is divided into two sections: the undirected fuzz testing baseline and the directed fuzz testing using Hermes.

We first established a baseline for the metrics specified in Sect. 4.2. This is achieved by executing the undirected fuzz testing method on the target application and logging both the code coverage and the number of mutations for the configured target area percentage.

Once a baseline is established, the evaluation of the directed fuzz testing method is executed with respect to the same metrics. Each directed evaluation is compared to its associated undirected counterpart for the given target area percentage. For example, the directed and undirected evaluations for the top 10 % of offending code are compared, and so on. The procedure steps are outlined in Fig. 2.

1. Execute undirected fuzz testing method on target application.
   (a) Configure method for target area percentage (10%, 20%, ...).
   (b) Exhaustively fuzz test target application with a full protocol (All features turned on).
   (c) Record metrics
2. Execute directed fuzz testing method on target application.
   (a) Configure method for target area percentage (10%, 20%, ...).
   (b) Generate targeted protocol using Hermes' genetic algorithm (3600-mutation "tracer round").
   (c) Exhaustively fuzz test target application using targeted protocol.
   (d) Record metrics.
3. Compare results.

Fig. 2. Evaluation procedure

"Offending code" in these evaluations are the lines of code contained in the method where a potential defect is found. For example, the top 10 % of the offending code will reflect the lines of code for each of the top 10 % most-severe potential defects – based on the FindBugs bugrank metric.

Our evaluation includes ten variations in the percentage of the offending code in increments of 10 % from 10 % to 100 %. We chose to include these increments

in an attempt to identify possible "sweet-spots" where coverage or performance would perform exceptionally-well compared to other results.

Both the undirected and directed methods were executed and their code coverage of the target areas measured. The protocol for the baseline calculation was kept constant (a full protocol with all features available) and EMMA was configured to track the code coverage for a given target area. This baseline code coverage was then compared to the calculated values of Hermes' directed fuzz approach.

Directed protocols were generated by Hermes' genetic algorithm with specific targets (in this case percentage of most severe offending code) in mind. The resulting best-fit protocol was then exhaustively evaluated to produce code coverage and number-of-mutation metrics for that target range.

4.1 Target Application

The target application selected for evaluation is the Crawler4J [31] library. To fuzz test this library a simple crawler was developed using Crawler4J and configured to connect to the Hermes fuzz test server.

The Crawler4J library was selected for evaluation because it is a Java-based application with its most recent revision being downloaded just under 15,000 times, it is designed to be extended, and it is open-source. The fact that Crawler4J is open-source allowed Hermes to provide full introspection and utilize its white-box features to their fullest extent.

4.2 Measurement

We designed our experiment to evaluate code coverage and performance. Code coverage was used as the fitness criteria for the genetic algorithm while performance was used to supplement the comparison of results. Thus performance was a secondary goal to maximizing code coverage of the target code.

Code Coverage. Code coverage is a measure of the amount of source code a specific test suite is able to evaluate.

There are three common forms of code coverage: function coverage, path coverage, and statement coverage [46]. Function coverage is the number of functions that are called by a given test suite. Statement coverage is the total number of individual statements executed by the test suite. Path coverage measures the coverage of all possible routes through the executed code. These values are compared to the total number of functions, statements, or paths in the target application to produce a code coverage percentage.

In addition the the general definition of code coverage, Sutton [6] defines code coverage within the context of fuzz testing to be "the amount of process state a fuzzer induces a target's process to reach and execute".

Code coverage was chosen as a metric for Hermes because, although it is "well-known that random testing usually provides low code coverage, and performs poorly overall [in that respect]" [13,14], code coverage has been extensively

used as a metric to measure the performance of fuzz testing [5,16,29,32,46–48], additionally, there is a general "lack of measurable parameters that describe fuzz test completeness" to draw from [29]. Specifically, we chose line coverage for the entire method where a target bug type was identified as the metric used for our analysis.

Performance. The evaluation of Hermes requires a performance metric. We chose to evaluate Hermes based on the number of mutations that the fuzzer generates to achieve a target code coverage. A baseline performance metric is set by executing an undirected and exhaustive fuzz test on the target while logging the number of mutations and code coverage achieved.

In the context of the Crawler4J crawler, the number of mutations equals the number of crawler requests to the server as each request included a single mutation of the protocol.

Previous research has utilized a variety of performance metrics including number of fuzzed inputs, total errors found, errors found per hour, and number of distinct errors found per hour [5,15,16,47].

The number of fuzzed inputs (mutations) was chosen because it is less subjective than a pure time comparison. A time comparison could be improved by simply increasing the CPU power of the host machine and would introduce a subjective aspect to our analysis.

4.3 Configuration

The default selection criteria for Hermes' analyser was set to use the FindBugs internal "bugrank" metric. Bugrank is calculated by FindBugs using a combination of the potential defect's category type and the type of potential defect found. The bugrank metric represents an overall severity metric for the potential defect.

The genetic algorithm was configured to be more aggressive in its mutation capabilities. This allows for more features to be brought back into the "gene pool" if there is an early, dominating, feature string that does not converge to a maximum later in the analysis, or if the population size is small. The initial configuration of the genetic algorithm is detailed in Fig. 3.

High growth ratios and increased mutation rates allow for quicker convergence in simple problems but suffer with more complex problems. These issues

P(Crossover) = 0.5
P(Mutation) = 0.05
Number of Generations = 30
Feature Strings per Generation = 10
Selection Algorithm = Tournament Selection (size=3)
Mutation Limit = 3600

Fig. 3. Initial configuration of Hermes' genetic algorithm

can be mitigated partially by increased population sizes and multiple populations with varying success [37]. In the configuration detailed in Fig. 3, a small population is used with a higher mutation, and lower crossover, probability. A higher population increases variability within the population, which will include more feature strings with high fitness, but it will slow the convergence to a maximum. For the purposes of time we chose a small population size of 10 for 30 generations. The mutation limit of 3600 was chosen to limit the amount of time required to evaluate a single feature string in the genetic algorithm. This initial mutation limit of 3600 will be used as a "tracer round" to target the full fuzz testing capabilities.

Typical crossover probabilities lie in the 0.5 to 1.0 range, while typical mutation probabilities are in the 0.005 to 0.05 range [41]. Tournament selection was used because "ranking and tournament selection are shown to maintain strong growth under normal conditions, while proportionate selection without scaling is shown to be less effective in keeping a steady pressure towards convergence" [37].

We configured Hermes to act as a "honeypot" server where it captures HTTP requests by providing non-repeating and self-directed links back to itself. Once an HTTP client, such as a crawler, is captured the server responds with mutated HTML mixed with valid HTML and begins the fuzz test process. The use of valid HTML within the server's response ensures that HTTP clients have a valid HTML link back to the server so that it may continue to be captured.

5 Results

5.1 Baseline: Undirected Fuzz Testing with Code Coverage

We calculated the baseline by executing a full-fuzz test (brute-force) on the full protocol definition. By brute-forcing the protocol we are able to produce the worst-case values for number of mutations, mean code coverage, the standard deviation for the mean code coverage, the mean code coverage of the targeted code's complement, and the standard deviation for the complement's mean code coverage. These results are detailed in Table 1. The mean code coverage is used with the standard deviation to provide an overall value for all of the sections of targeted code. The target code complement represents the code coverage that is *not* part of the target scope. The total number of possible unique mutations for the full target protocol is 67788. The baseline exhaustively evaluates the protocol which explains the constant number in the "Mutations" column.

The data in Table 1 follows an expected behaviour where the code coverage is high and the standard deviation is low when only looking at small sections of the code, but as the amount of target code increases so does the standard deviation and the mean coverage decreases.

For the baseline, we observed that a noticeable drop in code coverage and increase in standard deviation when the target code reaches 40 %. This may signal that a defect with little or no code coverage was added that was not in the previous set. In fact, observation of the detailed baseline results for 30 %

and 40 % showed that *two* defects are added with little code coverage. This would cause the drop in coverage we observed. The discrepancy between 30 % and 40 % is most prevalent in the standard deviation values where we observe a jump from a standard deviation of 0.1575 to 0.3606 – more than double.

Table 1. Baseline results from an undirected and exhaustive fuzz test with a full protocol

Targeted % of Code	Mutations	Mean Coverage	Standard Deviation	Complement Mean Coverage	Complement Std. Deviation
10	67788	0.8625	0.1304	0.4343	0.4664
20	67788	0.79	0.1485	0.4339	0.4663
30	67788	0.8425	0.1575	0.4334	0.4663
40	67788	0.7036	0.3606	0.4343	0.4664
50	67788	0.7283	0.3548	0.4337	0.4662
60	67788	0.785	0.3145	0.4337	0.4664
70	67788	0.8148	0.2901	0.433	0.4662
80	67788	0.7834	0.3248	0.4334	0.4663
90	67788	0.7493	0.3339	0.433	0.4663
100	67788	0.7415	0.3315	0.4328	0.4663

5.2 Directed Fuzz Testing with Code Coverage

Our evaluation analyzes the target application in two steps:

1. Find the best-fit candidate protocol that performs best under a restricted number of mutations – in this case 3600 mutations. This is the "tracer round" that directs the full-fuzz test evaluation. Table 2 details the results for each best-fit candidate.
2. Exhaustively fuzz test the best-fit protocol to fully evaluate the target application with respect to the given target code. Table 3 details the results of exhaustively fuzzing the best-fit protocols.

The results in Table 2 were surprisingly similar to the baseline detailed in Table 1 – with a significant reduction in mutations. Most of the mean coverage results for the best-fit protocols were within 3 % of their baseline counterparts. This is true for all values except the 80 % evaluation which differed by 5.58 %. After further investigation it was revealed that this drop in accuracy was due to a significantly-lower code coverage in a single section of offending code. As with the baseline, we observe the jump in standard deviation at the 30 % to 40 % mark. Finally, these values do not represent the entire best-fit protocol and

Table 2. Results from the best-fit candidates produced by Hermes

Targeted % of Code	Mutations	Mean Coverage	Standard Deviation	Complement Mean Coverage	Complement Std. Deviation
10	3600	0.8425	0.1622	0.4116	0.4624
20	3600	0.775	0.1636	0.412	0.4625
30	3600	0.8312	0.172	0.4116	0.4624
40	3600	0.6954	0.3623	0.4115	0.4624
50	3600	0.7208	0.3569	0.4115	0.4624
60	3600	0.752	0.3131	0.4094	0.4619
70	3600	0.8064	0.2962	0.4102	0.4622
80	3600	0.7276	0.3472	0.41	0.4621
90	3600	0.7441	0.3343	0.4097	0.4622
100	3600	0.7365	0.3318	0.4108	0.4624

they must be exhaustively evaluated to assure that the results are not simply "surface" matches.

The results from an exhaustive evaluation of the best-fit protocols are detailed in Table 3. Here, we observe fluctuations in the number of mutations, and thus the computation time, of the evaluations. This is the result of Hermes tailoring each protocol to attack the specified set of potential defects while pruning any redundant or useless features to minimize the total number of mutations. Furthermore, 8 of the 10 evaluations achieve parity with their baseline counterparts

Table 3. Results from exhaustively evaluating the generated best-fit protocols

Targeted % of Code	Mutations	Mean Coverage	Standard Deviation	Complement Mean Coverage	Complement Std. Deviation
10	41964	0.8625	0.1304	0.4171	0.4637
20	51648	0.79	0.146	0.4307	0.4656
30	41964	0.8425	0.1575	0.4347	0.4662
40	35508	0.7027	0.3607	0.4161	0.4635
50	51648	0.7283	0.3548	0.432	0.4657
60	50185	0.785	0.3145	0.4311	0.4658
70	41964	0.81	0.2939	0.4162	0.4638
80	23672	0.7834	0.3248	0.4147	0.4634
90	49162	0.7493	0.3339	0.4316	0.4659
100	49981	0.7415	0.3315	0.4319	0.4658

on mean code coverage, and the other 2 evaluations are within 0.5 % of *their* baseline counterparts.

The standard deviation results followed a similar trend set by the mean coverage. Seven of the ten evaluations achieved parity with their baseline counterparts with the other 3 evaluations within 0.4 % of their baseline counterparts. In one case, the 20 % evaluation, we observed a *decrease* in the standard deviation of the mean coverage while continuing to maintain mean coverage parity.

6 Analysis

6.1 Best-Fit Protocols and Their Accuracy

The best-fit protocols generated by Hermes were produced by selecting the best-performing protocol in a 3600-mutation evaluation (detailed in Table 2). The resulting protocols were then exhaustively evaluated (fuzzed) to produce the values shown in Table 3.

Our analysis compared the code coverage for the 3600-mutation best-fit protocol and its exhaustively-fuzzed counterpart. We observed that although the full evaluation achieves better code coverage in every evaluation it does not deviate from the initial best-fit evaluation with 3600 mutations in a significant manner. The two major discrepancies are at the 60 % and 80 % evaluations with a difference of 3.3 % and 5.58 % respectively. Additionally, the full evaluation achieves a lower or equivalent standard deviation compared to the initial evaluation. The notable anomalies are at the lower percentage targets (10 %, 20 %, and 30 %) and at the 80 % evaluation. At these areas we see a lower standard deviation than the initial best-fit evaluations.

Table 4. A comparison of the baseline and the full evaluations of best-fit protocols

Targeted % of Code	Difference in Mean Coverage (%)	Difference in Std Deviation	Baseline # Mutations	Full Best-Fit # Mutations	Difference in # Mutations
10	0	0	67788	41964	25824
20	0	0.0025	67788	51648	16140
30	0	0	67788	41964	25824
40	0.0009	0.0001	67788	35508	32280
50	0	0	67788	51648	16140
60	0	0	67788	50185	17603
70	0.0048	0.0038	67788	41964	25824
80	0	0	67788	23672	44116
90	0	0	67788	49162	18626
100	0	0	67788	49981	17807

The similarity between the initial and full best-fit evaluations may be due to the size of the target application or the size of the feature string used in the genetic algorithm. The deviation between the initial and full evaluations may increase with a change in either of these two factors.

6.2 Comparing Directed and Undirected Approaches

The directed (full best-fit) evaluations and the undirected (baseline) evaluations are compared in Table 4. In this table we observe that the full best-fit mean coverage and standard deviation results achieve near-parity with their baseline counterparts. Additionally, the areas that *did not* achieve parity were within 0.5 % of their targets.

We observe from Table 4 that *every* evaluation was able to reduce the number of mutations (and thus computation time as described in Sect. 4.2) required. In the case of the 80 % evaluation Hermes was able to reduce the number of mutations by 65 % from 67788 to 23672 mutations while achieving complete parity in both mean code coverage *and* standard deviation. The minimum improvement observed from our evaluations is a decrease in the number of mutations by 23.8 % (from 67788 to 51648 mutations) while maintaining parity within 0.5 % of mean code coverage and standard deviation.

7 Conclusion

Evidence in security assurance cases must be definitive, convincing, and accurate. The more specific the evidence the stronger the associated assurance argument. Fuzz testing is has become a popular tool for software security assurance but in its traditional (undirected) form, it provides only weak evidence for specific security cases. We have presented a method, tool implementation and experimental results for a directed fuzzer, which can be used to target specific potential code vulnerabilities. Our experimental results indicate that the method shows promise in reducing resources needed for covering code that is of interest from a security perspective (as indicated by static code analysis). We were able to achieve reductions in execution time (ranging from 23.8 % to 65 %), while targeting specific bug types (in this evaluation we chose the most-severe defects), and achieving near-equivalent code coverage to an exhaustive fuzz test (within 0.5 %) Clearly, our evaluation to date is limited, since we have only studied one real-world target software system. Additional experiments are needed to confirm the generalizability of our results.

References

1. Lipner, S.: The trustworthy computing security development lifecycle. In: 20th IEEE Computer Security Applications Conference, pp. 2–13. IEEE (2004)

2. Agudo, I., Vivas, J., Lopez, J.: Security assurance during the software development cycle. In: International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing, p. 20. ACM (2009)
3. Kelly, T., Weaver, R.: The goal structuring notation-a safety argument notation. In: Dependable Systems and Networks 2004 Workshop on Assurance Cases. Citeseer (2004)
4. Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: NDSS, vol. 8 (2008)
5. Takanen, A., Demott, J., Miller, C.: Fuzzing for software security testing and quality assurance. Artech House (2008)
6. Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Addison-Wesley Professional (2007)
7. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990)
8. DeMott, J.: The evolving art of fuzzing. Technical report, DEF CON, vol. 14 (2006)
9. Marshall, A., Howard, M., Bugher, G., et al.: Security best practices for developing windows azure applications. Technical report, Microsoft Corporation (2010)
10. Howard, M., Lipner, S.: The security development lifecycle, vol. 11. Microsoft Press (2009)
11. Goertzel, K.M., Winograd, T., McKinley, H.L., et al.: Software security assurance: a State-of-Art Report (SAR). DTIC Document (2007)
12. Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: IEEE Symposium on Security and Privacy (SP), pp. 497–512. IEEE (2010)
13. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. ACM Sigplan Not. **40**(6), 213–223 (2005)
14. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI **8**, 209–224 (2008)
15. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: IEEE 31st International Conference on Software Engineering, pp. 474–484. IEEE (2009)
16. Wu, Z., Atwood, J.W., Zhu, X.: A new fuzzing technique for software vulnerability mining. In: IEEE CONSEG, vol. 9. IEEE (2009)
17. Jain, L.C., Karr, C.L.: Introduction to evolutionary computing techniques. In: Electronic Technology Directions, pp. 122–127 (1995)
18. Holland, J.H.: Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. U. Michigan Press (1975)
19. Belew, R.K., McInerney, J., Schraudolph, N.N.: Evolving networks: using the genetic algorithm with connectionist learning. Citeseer (1990)
20. Whitley, D.: A genetic algorithm tutorial. Stat. Comput. **4**(2), 65–85 (1994)
21. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer, Berlin (2010)
22. Chess, B., McGraw, G.: Static analysis for security. Secur. Priv. **2**(6), 76–79 (2004). IEEE
23. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., et al.: Using static analysis to find bugs. Software **25**(5), 22–29 (2008). IEEE

24. Nagappan, N., Ball, T.: Static analysis tools as early indicators of pre-release defect density. In: ACM 27th International Conference on Software Engineering, pp. 580–586. ACM (2005)
25. Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. ACM SIGSOFT Softw. Eng. Not. **29**(6), 97–106 (2004). ACM
26. Ball, T.: The concept of dynamic analysis. In: Wang, J., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, pp. 216–234. Springer, Heidelberg (1999)
27. Mock, M.: Dynamic analysis from the bottom up. In: WODA 2003 ICSE Workshop on Dynamic Analysis, p. 13 (2003)
28. Ernst, M.D.: Static and dynamic analysis: synergy and duality. In: WODA 2003: ICSE Workshop on Dynamic Analysis, pp. 24–27 (2003)
29. Clarke, T.: Fuzzing for software vulnerability discovery. Department of Mathematic, Royal Holloway, University of London. Technical report. RHUL-MA-2009-4 (2009)
30. Yang, Q., Li, J.J., Weiss, D.M.: A survey of coverage-based testing tools. Comput. J. **52**(5), 589–597 (2005)
31. Crawler4j - Open Source Web Crawler for Java. https://github.com/yasserg/crawler4j
32. Oehlert, P.: Violating assumptions with fuzzing. IEEE Secur. Priv. **3**(2), 58–62 (2005)
33. Clarke, T., Crampton, J.: Fuzzing or how to help computers cope with the unexpected. Technical report, Royal Holloway University of London (2009)
34. Aitel, D.: The advantages of block-based protocol analysis for security testing. Technical report, Immunity Inc. (2002)
35. Juranic, L.: Using fuzzing to detect security vulnerabilities. Technical report, Infigo Information Security (2006)
36. Goodman, E.D.: Introduction to genetic algorithms. In: GECCO Conference Companion on Genetic and Evolutionary Computation, pp. 3205–3224. GECCO (2007)
37. Goldberg, D.E., Deb, K.: A comparative analysis of selection schemes used in genetic algorithms. In: Foundations of Genetic Algorithms, pp. 69–93 (1991)
38. Gen, M., Cheng, R.: Genetic Algorithms and Engineering Optimization. Wiley, New York (2000)
39. Deep, K., Mebrahtu, H.: Combined mutation operators of genetic algorithm for the travelling salesman problem. Int. J. Comb. Opt. Prob. Inf. **2**(3), 1–23 (2011)
40. Srinivas, M., Patnaik, L.M.: Genetic algorithms: a survey. IEEE Comput. **27**, 17–26 (1994). IEEE
41. Srinivas, M., Patnaik, L.M.: Adaptive probabilities of crossover and mutation in genetic algorithms. IEEE Trans. Syst. Man Cybern. **24**(4), 656–667 (1994). IEEE
42. Fortin, F., De Rainville, F., et al.: DEAP: evolutionary algorithms made easy. J. Mach. Learn. Res. **13**(1), 2171–2175 (2012)
43. Sulley: A Pure Python Fully-Automated and Unattended Fuzzing Framework. https://github.com/OpenRCE/sulley
44. Emma, A Free Java Code Coverage Tool. http://emma.sourceforge.net/
45. FindBugs - Find Bugs in Java Programs. http://findbugs.sourceforge.net/

46. Marovic, B., Wrzos, M., Lewandowski, M., et al.: GN3 quality assurance best practice guide 4.0. Technical report (2012)
47. Guang-Hong, L., Gang, W., Tao, Z., et al.: Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. In: Third International Conference on Convergence and Hybrid Information Technology, IEEE ICCIT 2008, vol. 2, pp. 491–497 (2008)
48. Iozzo, V.: 0-knowledge fuzzing. Technical report, Black Hat DC (2010)