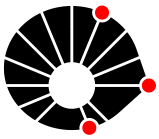

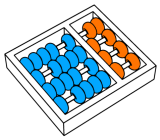

Testes de Segurança de Aplicações Usando Fuzzers

F. B. Silvério *E. Martins*

Relatório Técnico - IC-PFG-20-37
Projeto Final de Graduação
2020 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Testes de Segurança de Aplicações Usando Fuzzers

Flávia Bertoletti Silvério[1], Eliane Martins[2]

[1,2] Instituto de Computação Universidade Estadual de Campinas (UNICAMP), Caixa Postal 6176

13083-970 Campinas-SP, Brasil

Resumo. *Fuzz testing* é um método de testes que gera e insere no programa alvo entradas aleatórias com uma velocidade muito alta a fim de gerar e documentar *crashes*. Seu volume alto de entradas e sua abordagem automatizada têm como consequências achar *bugs* que escapam ao programador e testador.

O objetivo deste trabalho foi a análise do uso de *fuzzers* e sua aplicação em quatro estudos de caso escritos em linguagem C, encontrando-se dois *bugs* após campanhas de até 24 horas com as ferramentas AFL e honggfuzz.

Palavras-Chave: Segurança, Fuzz testing, Fuzzing.

1. Introdução

Seja pelas perdas financeiras acarretadas por *crashes* em software, ou pelo volume cada vez maior de informações importantes e privativas sendo transmitido todos os dias, a importância de termos sistemas de software seguros e robustos nunca foi maior do que nos dias atuais.

Neste contexto, o *fuzz testing* se apresenta como uma medida complementar aos tradicionais métodos de teste de software e análise do código, sendo uma alternativa automatizada para encontrar vulnerabilidades, que são defeitos (*bugs*) passíveis de exploração maliciosa.

De modo geral, o *fuzz testing* gera e insere no programa entradas aleatórias com uma velocidade muito alta a fim de gerar e documentar *crashes*. Seu volume alto de entradas e sua abordagem automatizada têm como consequências achar *bugs* que escapam ao programador e testador e que podem ser severos do ponto de vista da segurança, como vazamentos de memória.

O objetivo deste trabalho foi a análise do uso de *fuzzers* e sua aplicação em alguns estudos de caso. A preferência foi para a aplicações escritas em linguagem C, pois o objetivo é determinar os *fuzzers* a serem utilizados nos testes de sistemas embarcados.

Este trabalho dividiu-se em três principais passos: o primeiro foi dedicado à pesquisa sobre este conceito de teste, o funcionamento das ferramentas disponíveis e a escolha daquelas que mais se adequaram ao que foi proposto. No segundo, foram feitas pesquisas de bons programas-alvo para os quais utilizaram-se os dois *fuzzers grey-box*, honggfuzz e American Fuzzy Lop (AFL) em campanhas de até 24 horas a fim de encontrar vulnerabilidades em 4 programas com variados propósitos. No terceiro passo houve a análise dos resultados, em que se encontraram 2 bugs que resultaram em crashes em um programa.

Também na terceira parte houve a comparação das ferramentas utilizadas, determinando-se que para o contexto utilizado, o *fuzzer* AFL demonstrou melhores resultados quando avaliamos o número de bugs encontrados.

2. Fuzzing

2.1 Objetivos e passos

Embora o conceito de *fuzzing* seja simples, diversas etapas são necessárias para que o processo seja bem sucedido e possa atingir seu objetivo principal de encontrar falhas de segurança através dos testes do programa com entradas randômicas.

Com base neste conceito, podemos dividir o fluxo de execução em três partes: o pré-processo – atividades que acontecem antes da execução de determinada entrada-, a execução, e a atualização da configuração.

2.1.1 Pré-processo

As seguintes atividades fazem parte do pré-processo, embora a presença de algumas varie dependendo dos parâmetros da campanha e ferramentas escolhidas:

- **Instrumentação do programa:** refere-se à preparação do programa para que determinados parâmetros possam ser monitorados durante a campanha de *fuzzing*. A instrumentação pode acontecer de forma dinâmica (durante a execução) ou estática (no momento da compilação ou instrumentação do binário com auxílio de ferramentas específicas para este fim). Esta última normalmente tem menor impacto no tempo de execução, embora a instrumentação dinâmica possibilite que o teste atinja também bibliotecas dinamicamente ligadas ao programa alvo [1].

- **Seleção de sementes (*seeds*):** as sementes são escolhidas como entradas iniciais com base nas quais o *fuzzer* irá gerar as próximas entradas. A ideia é que este conjunto seja o menor possível, com casos de teste pequenos e que possam atingir diversas áreas do programa alvo. O *fuzzer* American Fuzzy Lop, por exemplo, pede em sua documentação que as entradas iniciais sejam, de preferência, menores que 1Kb e que só existam múltiplos arquivos se eles forem funcionalmente diferentes entre si. Em estudos mais aprofundados, pode-se procurar um conjunto ótimo com testes pequenos e cuja escolha maximize a porcentagem de código atingido pelas execuções.

- Criação de uma aplicação *harness*: bastante útil quando existe uma função alvo a ser testada, a *harness* existe como um programa direcionador, cujo objetivo é chamar as funções ou áreas de interesse do código alvo para direcionamento da campanha.

- Geração de inputs: a partir das sementes selecionadas (e das configurações do *fuzzer*), criam-se novas entradas para o programa. Mais detalhes na seção 2.2.

2.1.2 Execução

A execução refere-se à etapa do uso dos casos de teste gerados no pré-processo como entradas do programa alvo. Nesta etapa, a velocidade com que o *fuzzer* executa cada interação é determinante para o sucesso dos testes, já que pela natureza randômica das entradas geradas, a maioria é rejeitada pelo programa sem atingir pontos chave que podem ser origens de *bugs*. Desta forma, é de extrema importância aumentar o número de execuções em um determinado período a fim de aumentar as chances de atingir partes críticas do alvo. Para isso, alguns *fuzzers* apresentam opções de otimização que visam aumentar a velocidade utilizando o contexto de *in-memory fuzzing*, que consiste em tirar uma “foto” do programa após o processamento inicial e restaurar o programa para esse estado a cada iteração, assim reduzindo o tempo de execução de cada entrada [1].

Além de maximizar o número de execuções, o *fuzzer* tem que identificar se as iterações resultam em *crashes*, e se um bug foi encontrado. Para isso é necessário o uso de um oráculo. Um oráculo de teste, ou simplesmente oráculo, é um mecanismo que determina se a saída fornecida pelo programa em teste está de acordo com o esperado [9]. Distinguir a resposta correta da incorreta para cada entrada é uma dificuldade, especialmente para oráculos automatizados. Esta dificuldade é comum aos diferentes tipos de testes e é conhecida como problema do oráculo.

A forma mais simples de oráculo em *fuzzers* consiste em distinguir as execuções que dão *crashes* daquelas que não dão. Este oráculo é simples pois prescinde da especificação, além do que, *crashes* são facilmente identificáveis. No entanto, o fato de não ter ocorrido um *crash* não significa que o programa executa corretamente. Para melhorar o potencial do

oráculo na descoberta de *bugs* pode-se utilizar sanitizantes (*sanitizers*), que são assertivas inseridas no código que levantam exceções quando um erro é detectado.

O sanitizante mais conhecido é o ASAN (Address Sanitizer), que aponta erros de acesso à memória, mas também existem sanitizantes para detectar comportamento indefinido (como UBSan), *casting* incorreto (TypeSan), e outras funcionalidades. Apesar de muito úteis, os sanitizantes geram um aumento do tempo de execução considerável, diminuindo sua viabilidade de uso para campanhas curtas de testes - o Memory Sanitizer (MSan), faz com que as execuções possam demorar o triplo do tempo original, por exemplo [6].

2.1.3 Atualização da configuração

Como os *fuzzers* geram evidência da falha – isto é, para cada *crash* encontrado, existe a entrada correspondente – é possível fazer uma triagem dos resultados gerados pelo oráculo.

A primeira parte da triagem é a deduplicação, ou seja, identificação dos casos em que várias entradas geraram *crashes* causados pelo mesmo bug. As razões para isso são otimizar o uso de memória e também ter uma melhor noção de quantos *bugs* realmente existem no programa testado. A deduplicação é mais comumente feita através de rastreamento de estado da pilha utilizando-se *hashing*, mas também pode ser feita com base na análise de cobertura do programa, ou através de uma análise semântica do fluxo de dados [1].

Os resultados da deduplicação podem ser usados para alimentar algoritmos evolutivos que gerarão novas entradas e atualizar as configurações do *fuzze*r a fim de, por exemplo, usar a informação sobre partes do código atingidas para maximizar o grau de cobertura da campanha.

2.2 Técnicas para geração de entradas

A qualidade das entradas geradas tem interferência direta nos resultados encontrados pela campanha de *fuzzing*. As técnicas utilizadas para essa etapa dependem da ferramenta,

porém, elas podem ser divididas em dois grupos: os *fuzzers* que geram a entrada baseados em modelo, e os que geram a entrada baseados em mutação.

Os *fuzzers* que geram a entrada baseados em modelo usam descrições das entradas aceitas pelo programa alvo e então, com base nisso, geram as novas entradas utilizadas na campanha. De modo geral, essas ferramentas dependem que o usuário as alimente com essas descrições de modelo, sendo que a gramática e formato destas descrições depende do *fuzzer* utilizado. Ainda nos *fuzzers* baseados em modelo, existem, mais raramente, também aqueles que inferem o modelo das entradas, e, portanto, não precisam que esta descrição seja providenciada pelo usuário. Esta inferência pode acontecer tanto no pré-processamento quanto na atualização da configuração [1].

Aqueles baseados em mutação, por outro lado, mudam pequenas partes das entradas providas pelo usuário a fim de gerar entradas randômicas que ainda mantenham parte da estrutura daquelas esperadas - maximizando as chances de que o que foi gerado seja aceito pelo programa - mas que seja randômico o suficiente para cumprir os propósitos da campanha de *fuzzing*. Nestes casos, várias técnicas de mutações podem ser utilizadas, a depender da ferramenta escolhida, tais como mutação aritmética (adicionar um pequeno valor em uma sequência de bits do input), *bit-flipping* (troca do valor de certa quantidade de bits), mutação baseada em dicionário (usando valores mais prováveis de serem gatilhos de *crashes*, como 0 ou -1), ou mutação baseada em blocos (deletando, substituindo ou adicionando um bloco a uma semente).

De modo geral, a geração de um modelo demanda mais recursos do que a utilização de um *fuzzer* baseado em mutação, já que o modelo depende de uma análise profunda da sintaxe e da semântica das entradas do programa. Por outro lado, os modelos garantem a geração de mais casos válidos, capazes de respeitar a semântica das entradas e manter dependência de valores entre campos, se necessário [2].

Nos dois casos, é importante que o *fuzzer* consiga manter equilíbrio entre revisitar as mesmas partes de um código com diferentes entradas a fim de tentar encontrar *crashes* e explorar novas áreas, tudo isso mantendo o custo de geração de cada entrada baixo, já que as iterações devem acontecer de maneira muito rápida.

2.3 Desafios

Apesar do conceito de fácil entendimento, os testes *fuzzing* acabam encontrando diversos desafios para gerar resultados. O primeiro desafio é encontrar bons alvos, já que há linguagens mais propensas a receberem esse tipo de teste (como C/C++) e linguagens com muito menos suporte e ferramentas disponíveis para que recebam *fuzz testing* (como Java, Python e Javascript).

O segundo desafio é encontrado na questão de instrumentação. Embora os vários tipos de instrumentação tornem possível o monitoramento e descoberta de diversos tipos de *bugs*, o acréscimo de tempo de execução pode fazer com que eles se tornem inviáveis para testes com tempo reduzido (algumas horas ou um dia) e gerem muito mais *timeouts* em programas complexos que provavelmente não são resultado de *loops* infinitos no alvo, mas sim da demora nas execuções causadas pelas funções de instrumentação.

O terceiro desafio a ser mencionado é que muitos *bugs* não são apontados pelos *fuzzers*, mesmo que o *fuzzer* atinja aquela parte específica do código. Portanto, um *fuzzer* que faz uma cobertura de 100% do código e não aponta nenhum *crash* não garante a inexistência de *bugs*, já que um acesso indevido de memória que aponta para uma posição que não está disponível na execução, mas que é adjacente e válida pode não gerar um *crash* [1], mas isso continua sendo uma falha de segurança que pode causar o vazamento ou perda de informações importantes. Por isso, *fuzz testing* pode ser usado como uma técnica complementar de testar programas quanto à sua robustez e segurança, mas não é indicado como técnica única.

2.4 Ferramentas existentes

Os *fuzzers* podem ser divididos em três grupos, de acordo com o conhecimento que o *fuzzer* tem do código fonte durante sua execução.

Fuzzers black-box, como o nome sugere, enxergam o programa alvo como uma caixa preta: eles não têm conhecimento do código fonte e seu fluxo de execução, e, portanto, tomam decisões baseadas apenas nas entradas, saídas e tempo decorrido por iteração. Sem o auxílio da instrumentação para guiar a campanha com base na cobertura de código, esse tipo de *fuzzer* pode ficar preso em testes muito parecidos com as sementes usadas na configuração, mas existem, no entanto, casos em que ferramentas *black-box* podem ser uma boa alternativa. Nesta lista estão os casos em que o resultado não é determinístico (ou seja, a mesma entrada pode gerar diferentes saídas), ou quando o alvo é muito lento ou muito grande, já que testes *black-box* podem ser paralelizados com maior facilidade.

Por outro lado, *fuzzers white-box* são aqueles que têm acesso completo ao programa e podem mapear seus fluxos de execução, muitas vezes através do que é chamado de *dynamic symbolic execution*. Esse tipo de execução mapeia as possíveis entradas com valores simbólicos e para cada condicional ou mudança de fluxo, os novos caminhos recebem novos símbolos, de modo que ao fim desse processo todos os caminhos presentes no programa são mapeados e o *fuzzer* pode usar métodos matemáticos para chegar a soluções que gerem entradas para atingir todos os fluxos possíveis [1], obtendo, em teoria, máxima cobertura e encontrando *bugs* que estão em lugares com acesso mais difícil e raro. Esses benefícios, entretanto, vêm com um grande custo: esse modo de execução se torna muito mais complexo do que o modelo *black-box*, fazendo com que o custo operacional seja muitas vezes inviável [3].

Desta forma, os fuzzers *grey-box* são uma tentativa de englobar os benefícios dos *fuzzers black* e *white-box*, sem que suas restrições tenham tanto impacto na campanha de testes. Essas ferramentas têm acesso a mais informações sobre o programa alvo que os *fuzzers black-box*, e, portanto, podem tomar decisões mais informadas sobre a geração de novas entradas e a atualização de configurações. A instrumentação utilizada por esse tipo de *fuzzer* é mais leve que a análise profunda das ferramentas *white-box*, reduzindo seu impacto no processamento de execução, mas ainda sendo capaz de gerar informações suficientes

para que o *fuzzer* possa entender quando atingiu uma nova área do alvo e orientar seus esforços para maximizar sua cobertura e encontrar *bugs* em áreas cujo acesso é mais remoto que as atingidas, à princípio, pelos *black-box*.

3 Seleção da ferramenta

3.1 Benchmarks utilizados

Entre todas as possibilidades de *fuzzer* citados na seção 2.4, foi decidido que as ferramentas selecionadas para os testes neste trabalho deveriam obedecer às duas seguintes características:

- ***Fuzzer grey-box:*** pelo tempo reduzido de testes, o uso de um *fuzzer white-box* se tornaria inviável pelos recursos necessários, então o *fuzzer grey-box* se mostrou como uma opção razoável em que se é possível encontrar resultados de interesse em campanhas curtas, mas ainda utilizando um nível de inteligência quanto ao direcionamento de esforços computacionais.

- ***Fuzzer* baseado em mutação:** essa característica foi escolhida tanto pela maior facilidade de uso quanto pela popularidade e número de ferramentas que utilizam essa configuração, aumentando-se o leque de escolhas e de *benchmarks* que podem ser utilizados como fonte de informações.

Com base nestes, foram escolhidos dois *benchmarks* em que a escolha de ferramentas foi baseada.

O primeiro *benchmark* é o Magma [4], que avaliou seis *fuzzers grey-box* baseados em mutação durante 26000 horas de uso de CPU. Por princípio, esse *benchmark* utilizou programas reais e diversos e os *fuzzers* foram avaliados por números de *bugs* encontrados e seu tempo para encontrá-los.

O segundo *benchmark* utilizado foi o Fuzzbench [5], que é um serviço gratuito e fornecido pelo Google para disponibilizar uma plataforma de testes de *fuzzers* e seus relatórios de resultados. O Fuzzbench utiliza normalmente dez *fuzzers* e 24 programas alvo diversos para

seus relatórios, e sua forma de comparação é através de cobertura atingida, uma métrica que não consta no relatório gerado pelo *benchmark* Magma.

3.2 Critérios de seleção

O primeiro passo da seleção das ferramentas consistiu em utilizar os *benchmarks* Magma e Fuzzbench para alguns fuzzers grey-box baseados em mutação.

O Fuzzbench começou a divulgar em junho de 2020 relatórios com notas de comparação entre as estatísticas de cobertura (nota normalizada, quanto mais próximo de 100 melhor o desempenho do *fuzzer* neste aspecto). Foram tabulados os resultados de todos os relatórios divulgados entre os meses de junho e setembro de 2020, gerando a tabela a seguir.

Tabela 1- Média normalizada da cobertura alcançada pelos *fuzzers* no ano de 2020

Fuzzer	**12/06**	**03/08**	**16/08**	**23/08**	**31/08**	**07/09**	**18/09**	**24/09**	**28/09**	**Total**
honggfuzz	96,76	95,94	96,19	95,54	95,36	95,37	94,89	94,71	95,57	95,79
AFL++	89,56	91,00	96,83	96,62	-	-	96,39	97,40	96,14	94,63
AFL	90,19	91,21	91,20	90,79	90,23	89,89	89,81	90,02	89,99	90,30
moptafl	85,96	90,51	90,80	90,32	89,82	89,60	89,51	89,54	89,70	88,69
aflfast	87,04	88,66	89,25	88,22	87,69	87,67	87,58	87,61	88,01	87,90
fairfuzz	81,55	85,43	86,65	85,05	84,88	84,74	84,67	84,70	87,89	85,16

Como nota de corte, determinou-se a nota de 90 em relação à cobertura como mínimo para manter o *fuzzer* como candidato às ferramentas escolhidas, eliminando-se assim os *fuzzers* moptafl, aflfast e fairfuzz. A partir disso, a escolha final se deu com os resultados apresentados no *benchmark* Magma.

Neste, duas informações de interesse foram selecionadas, a primeira sendo o *fuzzer* com melhor desempenho para cada programa alvo. A Tabela 2 (que equivale à Tabela 3 da referência [4]) apresentada a seguir contém as informações coletadas.

Tabela 2- Número médio de bugs encontrados e desvio padrão por *fuzzer* durante dez campanhas. O *fuzzer* de melhor desempenho por programa alvo recebe realce verde.

Target	honggfuzz	afl	moptafl	aflfast	afl++	fairfuzz
libpng	3.6 ± 0.52	1.6 ± 0.70	1.0 ± 0.00	1.3 ± 0.48	1.2 ± 0.42	1.5 ± 0.53
libtiff	4.7 ± 0.67	4.8 ± 0.42	4.4 ± 0.52	4.2 ± 0.42	3.4 ± 0.52	4.8 ± 1.23
libxml2	5.0 ± 0.47	2.9 ± 0.32	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	2.6 ± 0.52
openssl	2.8 ± 0.63	2.9 ± 0.57	3.0 ± 0.00	2.6 ± 0.52	3.0 ± 0.67	2.0 ± 0.47
php	2.8 ± 0.42	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	1.6 ± 0.70
poppler	3.8 ± 0.92	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	2.9 ± 0.32
sqlite3	3.2 ± 0.63	0.8 ± 0.92	1.2 ± 1.14	1.2 ± 0.79	1.6 ± 0.84	1.5 ± 0.53

Com base nessa tabela, vemos que entre os três *fuzzers* selecionados quanto à cobertura, o honggfuzz aparece como aquele com melhor performance em um maior número de programas alvo; entretanto, os outros dois candidatos (AFL e AFL++) se destacaram em dois programas alvo cada.

A segunda informação de interesse é o tempo que cada um dos *fuzzers* levou para apontar cada um dos bugs encontrados. Com base nesta informação, o número de *bugs* encontrados por ferramenta foi contabilizado, criando-se a Tabela 3.

Tabela 3- Número de *bugs* encontrados, com base Tabela 4 da Referência [4]

Fuzzer	Número de bugs encontrados
honggfuzz	33
AFL	24
AFL++	23
fairfuzz	23
moptafl	22
aflfast	22

Nela, vemos que entre as ferramentas pré-selecionadas, aquelas que aparecem com melhor resultado foram honggfuzz e AFL. Com base nisto e nas comparações anteriores, estes *fuzzers* foram selecionados para a continuidade deste trabalho.

3.3 Ferramentas selecionadas

Com base nos critérios anteriores, os dois *fuzzers grey-box* baseados em mutação que foram selecionados para os testes foram o American Fuzzy Lop (AFL) [7] e o honggfuzz [8].

O American Fuzzy Lop é uma ferramenta bastante popular e com interface amigável ao usuário, funcionando através de uma instrumentação com baixo impacto na performance que acontece no momento de compilação. A partir desta instrumentação, o *fuzzer* guia sua campanha determinando se a entrada gerada atingiu um novo estado no programa, e se sim, a adiciona no conjunto dinâmico de sementes (chamado corpus), reduzindo-a ao menor tamanho possível que mantenha suas características de cobertura. O AFL também usa uma combinação das técnicas de mutação para gerar suas entradas – como *bit-flipping*, mutação aritmética e baseada em dicionário-, e um de seus princípios é a confiabilidade: nenhuma nova *feature* que cause instabilidade na execução e possa causar *crashes* abruptas ao usuário é bem-vinda.

O *fuzzer* honggfuzz, assim como o AFL, é uma ferramenta bastante popular. Por padrão, tem características multi-thread e multi-processo, usando todo o potencial disponível pela CPU. Também é um *fuzzer* inteligente, identificando quando uma nova área do código é atingida e adicionando a entrada ao corpus para sofrer novas mutações, com a possibilidade de minimização deste conjunto, assim como o AFL. Ambos os *fuzzers* tem modos de campanha chamados "persistentes", utilizando-se o *in-memory fuzzing* e diminuindo o tempo de cada iteração.

4 Estudos de caso e resultados

4.1 Descrição dos experimentos

Neste relatório, foram usados quatro estudos de caso variados em campanhas de até 24 horas que aconteceram em ambiente Linux, com ambas as ferramentas selecionadas. As campanhas utilizaram, em suas respectivas seções, a menos que especificado o contrário, as configurações padrão de ambos os *fuzzers* – comparando-se assim a forma com que atingem seus resultados com as configurações de entrada disponíveis em suas

documentações. Os programas utilizados são escritos primariamente em C, por sua afinidade com este tipo de análise, e compilados com os compiladores das respectivas ferramentas – *afl-gcc* no caso do AFL e *hfuzz-clang* no caso do honggfuzz.

Quando o próprio projeto disponibilizou em seu repositório entradas variadas para teste, estas eram utilizadas no conjunto de sementes. Caso contrário, era criado um arquivo em formato aceito pelo programa, e este era tido como entrada inicial. O conjunto de sementes e ordem deste foram os mesmos para ambas as ferramentas.

Consideraram-se aqui resultados relevantes o número de *bugs* apontados além do número de execuções obtidas no modo de execução baseado em *feedback (grey-box*), uma vez que a velocidade de execução é imprescindível para o próprio conceito destas ferramentas. Como observação adicional temos o tamanho do corpus – conjunto dinâmico de sementes alterado durante a execução conforme os *feedbacks* recebidos – do *fuzzer* honggfuzz ao final da campanha.

As subseções a seguir descrevem os estudos de caso utilizados nos testes.

4.2 Capstone [10]

Capstone é uma ferramenta de desmontagem de binários para sua análise e engenharia reversa, funcionando em diversas plataformas de hardware. Sua versatilidade e arquitetura que prioriza baixo uso de memória e processamento são os atrativos em que o projeto se baseia para buscar sua posição de destaque na comunidade de segurança.

As campanhas com esta ferramenta duraram 24 horas para cada um dos *fuzzers*, sendo que neste período, não houve nenhum *bug* encontrado e reportado por nenhuma das ferramentas na versão 4.0.2 do software.

O *fuzzer* AFL obteve, nestas 24 horas, 79,7 milhões de execuções, enquanto o *fuzzer* honggfuzz obteve 51,2 milhões, o que coloca o American Fuzzy Lop como 56% mais rápido que seu concorrente neste estudo de caso.

O conjunto dinâmico de sementes se manteve como um arquivo para o *fuzzer* honggfuzz durante esta campanha.

4.3 Sed [11]

Esta ferramenta de análise e edição de textos em sistemas Unix é muito conhecida e utilizada principalmente pela sua praticidade em encontrar e editar expressões em arquivos sem precisar nem mesmo abri-los, e também pelas possibilidades de uso de expressões regulares para encontrar os padrões desejados.

As campanhas com o Sed duraram 18h, e enquanto ambas as ferramentas não encontraram nenhum *bug* na versão 1.0, o *fuzzer* honggfuzz conseguiu gerar e executar entradas com muito mais rapidez que a outra ferramenta estudada, sendo que o primeiro teve 50,9 milhões de iterações no período contra 20,6 milhões de vezes executadas pelo AFL, sendo, portanto, cerca de 147% mais rápido.

Esta campanha terminou com o corpus do *fuzzer* honggfuzz diminuindo de 395 para apenas um arquivo. Isso acontece pela funcionalidade de minimização de corpus, que exclui sementes funcionalmente redundantes.

4.4 Radare2 [12]

Radare2 é um projeto que visa a reescrita do software Radare, cujas funcionalidades englobam desde ser um editor hexadecimal e *debugger*, até funções para desmontagem e análise de binários, comparação, visualização e substituição de dados.

Neste software, o *fuzzer* American Fuzzy Lop encontrou dois diferentes *bugs* durante sua campanha que durou 24 horas e teve 1,46 milhões de execuções. Quando utilizado pelo mesmo tempo e com suas configurações padrão, o *fuzzer* honggfuzz não encontrou nenhum *bug* neste software em 1,56 milhões de execuções, mantendo-se o corpus com um arquivo.

Sabendo que o programa alvo tinha *bugs* passíveis de serem encontrados em uma campanha relativamente curta, foram feitos outros testes das funcionalidades do honggfuzz a fim de descobrir se, neste estudo de caso, mudanças em suas configurações poderiam

obter os mesmos resultados mostrados pelo AFL. Desta forma, duas novas campanhas de 24 horas foram realizadas com este *fuzzer*.

A primeira foi realizada ativando-se o modo *in-memory fuzzing*, chamado “persistente”, que tende a ter execuções mais rápidas. Este modo realmente obteve mais execuções durante a duração da campanha (16,38 milhões), mas ainda assim não revelou a presença de nenhum *bug*. Neste modo, o tamanho do corpus se manteve o mesmo durante toda a campanha.

A segunda campanha utilizou-se de uma compilação mais instrumentada (com a *flag fsanitize=address* na compilação), a fim de guiar com mais propriedade a execução do *fuzzer*. Este modo aumentou muito o tempo de execução de cada entrada, de modo que apenas 158 mil execuções foram realizadas neste período e novamente nenhum bug foi identificado. O tamanho do conjunto de sementes foi de uma para 1194 ao decorrer das 24 horas.

4.5 CANOpen [13]

Este protocolo baseado no modelo produtor/consumidor está presente em sistemas distribuídos de controle utilizados em automação, tais como equipamentos médicos, aplicações marítimas e ferroviárias. Esta aplicação preza pela compatibilidade com outras aplicações CAN por ser altamente padronizada.

O formato desta aplicação baseia-se numa execução contínua, em que o programa se mantém em um *loop* de recebimento de entradas e execução até que seja manualmente encerrado.

Este formato específico não é compatível com as ferramentas escolhidas, já que todas as execuções superam os limites de tempo propostos e geram *timeouts*, mesmo que bem sucedidas, fazendo com que o conjunto de sementes inicial não seja aceito como modelo de execução.

Desta forma, para que fosse possível testar a aplicação para falhas de segurança através do *fuzzing*, foi necessário que este aspecto dela fosse modificado, editando-se o código fonte

para que encerrasse o programa após o recebimento de uma entrada e execução correspondente.

Esta modificação da versão 1.3 foi testada durante 24 horas, nas quais o American Fuzzy Lop executou 151 milhões de iterações e o *fuzzer* honggfuzz apenas 73,7 milhões, tornando o AFL cerca de 105% mais rápido nesta aplicação, embora nenhum dos dois tenha encontrado *bugs* no tempo definido de campanha.

O tamanho de corpus para o *fuzzer* honggfuzz, que começou com um arquivo, terminou a campanha com oito sementes diferentes.

4.6 Ameaças à validade dos experimentos

Alguns aspectos podem ser vistos como ameaças à validade dos experimentos anteriores. O primeiro é que, por restrição de recursos, muitas campanhas foram feitas em paralelo, de forma que a capacidade computacional entre campanhas pode ter sido afetada.

O segundo ponto a ser considerado é que a escolha inicial de sementes pode não ter sido ótima em relação ao funcionamento do programa, já que não houve uma análise específica dos códigos fonte de forma a aumentar as áreas atingidas pelo conjunto de sementes escolhido.

O terceiro ponto acontece no estudo específico do protocolo CANOpen e as alterações que tiveram que ser feitas no código para que o teste *fuzzing* fosse possível: embora seu objetivo tenha sido retirar apenas o *loop* contínuo original presente no programa, não se pode afirmar com certeza que esta alteração não afetou os resultados dos testes.

5 Conclusões e trabalhos futuros

Devido à pouca disponibilidade de recursos, as campanhas realizadas foram curtas para programas que são complexos e que já têm comunidades ativas que realizam testes há vários anos, de forma que é compreensível que a maioria dos testes não tenha encontrado *bugs* nos alvos indicados. Desta forma, é sugerido para que em trabalhos futuros sejam realizadas campanhas de duração mais expressiva – uma semana ou mais - para os programas alvo escolhidos.

Especificamente para o programa CANOpen, devido a sua complexidade e importância de uso, seria interessante dedicar recursos para realizar testes específicos com um *fuzzer* baseado em modelo. Embora a análise do código para geração deste modelo seja custosa, este direcionamento pode fazer com que mais entradas atinjam áreas de interesse da aplicação, tornando os resultados da campanha mais completos e substanciais.

Por fim, como mostrado no experimento com a aplicação Radare2, nem sempre o número de execuções equivale ao maior número de *bugs* encontrados, já que a qualidade das entradas e do oráculo têm papel significante nestes resultados. A comparação entre os resultados dos *fuzzers* no Fuzzbench e Magma também mostra que nem sempre uma maior cobertura é diretamente proporcional às falhas apontadas. Desta forma, para programas alvo cujos *bugs* já estão documentados, uma análise estatística se torna atrativa para que se quantifiquem os impactos que as diversas variáveis de uma campanha têm em encontrar defeitos. Neste caso pode-se determinar, para determinado alvo, qual a relação entre custo (computacional e horas analíticas) e retorno para aspectos como formação de um conjunto de sementes inicial ótimo, uso de determinados tipos de sanitizantes e aplicação de modelos para as entradas.

6 Referências bibliográficas

[1] Manes VJM, Han HS, Han C, Cha SK, Egele M, Schwartz EJ, et al. Fuzzing: Art, Science, and Engineering [Internet]. Dezembro/2018.

[2] Payer M. The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes [Internet]. Janeiro-Fevereiro/2019.

[3] Godefroid P, Levin MY, Molnar D. Automated Whitebox Fuzz Testing [Internet]. Janeiro/2008.

[4] Hazimeh A, Payer M. Magma: A Ground-Truth Fuzzing Benchmark [Internet]. 2020. Disponível em: https://hexhive.epfl.ch/magma/docs/preprint.pdf

[5] Fuzzer Benchmarking As a Service. FuzzBench. Disponível em: https://google.github.io/fuzzbench/

[6] Clang 12 documentation. MemorySanitizer - Clang 12 documentation. Disponível em: https://clang.llvm.org/docs/MemorySanitizer.html

[7] Zalewski M. American Fuzzy Lop. GitHub - google/AFL. Disponível em: https://github.com/google/AFL

[8] honggfuzz. GitHub - google/honggfuzz. Disponível em: https://github.com/google/honggfuzz/

[9] Howden WE. In: Theoretical and empirical studies of program testing. Victoria, B.C.: University of Victoria, Dept. of Mathematics; Julho/1978. p. 293–8.

[10] Capstone. The Ultimate Disassembly Framework. Disponível em: http://www.capstone-engine.org/

[11] GNU sed. Disponível em: https://www.gnu.org/software/sed/

[12] Libre and Portable Reverse Engineering Framework. radare2. Disponível em: https://rada.re/n/

[13] CAN in Automation (CiA). CAN in Automation (CiA): CANopen. Disponível em: https://www.can-cia.org/canopen/