Geração de Testes de Software para Verificação de Faltas e Funcionalidades

Francisco Araújo, Ibéria Medeiros, and Nuno Neves

LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
fc45701@fc.ul.pt, imedeiros@di.fc.ul.pt and nuno@di.fc.ul.pt

Resumo O aumento da complexidade do software empregue em produtos industriais está diretamente relacionada com o crescimento exponencial do número de funcionalidades presentes nestes, introduzidas com vista a responder às exigências do mercado. Essa complexidade cresce quando é necessário criar variantes das aplicações a partir de diversos componentes de software, como acontece em sistemas embebidos. Tal complexidade dificulta o teste e a validação do software para os seus requisitos, bem como pode originar vulnerabilidades de segurança. Programas de software industrial necessitam de ser testados adequadamente, em termos de funcionalidades e de segurança, por forma a garantir elevados níveis de qualidade. Embora já existam ferramentas automáticas de validação de segurança, não existe nenhuma ferramenta que possibilite a reutilização de resultados de testes entre versões de aplicações, de modo a validar estas versões e variantes da maneira mais eficiente possível. Este trabalho propõe uma abordagem que permite testar variantes ainda não testadas a partir de resultados das que já foram avaliadas. A abordagem foi implementada numa ferramenta com base na aplicação de fuzzing American Fuzzy Lop (AFL) e foi validada com um conjunto de programas de diferentes versões. Os resultados experimentais mostraram que a ferramenta consegue obter melhores resultados que o AFL.

Keywords: Fuzzing · Detecção de Vulnerabilidades · Testes de Cobertura · Testes de Software · Segurança no Software.

1 Introdução

O rápido crescimento da complexidade de software unido com a grande necessidade de software no dia a dia causou uma exigência para testar os mesmos de modo a conseguir garantir um certo nível de qualidade, funcionamento e segurança. Por exemplo, tanto o carro que conduzimos hoje como o frigorifico que usamos para manter a temperatura desejada dos nossos alimentos, requer software de tal complexidade que quando postos sobre alto stress, poderiam apresentar algum tipo de bug. No caso desse bug ser uma vulnerabilidade, e, por conseguinte, poder ser explorada, seria capaz de por vidas em perigo e mesmo causar danos financeiros no valor de milhões de euros. Essa vulnerabilidade conseguiria, por exemplo, criar a hipótese ao atacante de tomar controlo do carro

ou, no caso do frigorifico, aumentar a temperatura fazendo com que a comida se estrague. Não obstante a isso, depois de essas vulnerabilidades terem sido descobertas, é necessário iniciar um processo de correção do software, custando tempo e dinheiro.

A complexidade do software cresce quando é necessário criar variantes das aplicações a partir de diversos componentes de software, como acontece em sistemas embebidos. Tal complexidade dificulta o teste e a validação do software para as funcionalidades que foi desenhado, bem como pode originar vulnerabilidades de segurança. As vulnerabilidades podem permanecer ocultas durante vários anos em qualquer programa, independentemente de quantos testes foram executados para tentar assegurar a sua qualidade e segurança. Isto é tanto devido à eficiência destes testes que podem ser de uma qualidade limitada, como devido ao curto tempo disponível para garantir a correta funcionalidade. Enquanto que um atacante externo possui tempo teoricamente ilimitado quando o software já se encontra no mercado.

Os atacantes externos só necessitam de encontrar uma única vulnerabilidade para conseguir tomar partido do software em si, enquanto que os testes desenvolvidos têm de encontrar inúmeros. Como resultado disto, hoje em dia as companhias gastam recursos em termos de custo e de tempo para conseguir melhorar o processo de verificação e validação de software, por forma a tentar garantir o nível de qualidade e segurança desejado em qualquer dos seus produtos. No entanto, como acima referido, os recursos e tempo são limitados nos testes, fazendo com que vários bugs e vulnerabilidades não sejam detetadas por estes testes, mantendo-se ainda nos produtos finais. Embora já existam ferramentas automáticas de validação de segurança, não existe nenhuma ferramenta que possibilite a reutilização de resultados de testes entre versões de aplicações, de modo a validar estas versões e variantes da maneira mais eficiente possível.

Este trabalho propõe uma abordagem que permite testar variantes ainda não testadas a partir de resultados das que já foram avaliadas. A abordagem foi implementada na ferramenta PandoraFuzzer, a qual tem por base a aplicação de fuzzing American Fuzzy Lop (AFL), e foi validada com um conjunto de programas de diferentes versões. Os resultados experimentais mostraram que a ferramenta melhora os resultados do AFL.

As contribuições deste trabalho são: (1) uma solução que permite a deteção de vulnerabilidades de modo mais eficiente, usando informação gerada em processos de testes anteriores; (2) a ferramenta PandoraFuzzer que implementa a solução proposta; (3) uma avaliação experimental com aplicações do *binutils*.

O resto do artigo está estruturado seções seguintes: a Secção 2 apresenta as técnicas de fuzzing onde assenta este trabalho. A Secção 3 apresenta a arquitectura da solução proposta, enquanto a Secção 4 o protótipo que a implementa e a Secção 5 a sua avaliação. A Secção 6 apresenta o trabalho relacionado e a Secção 7 concluí o artigo.

2 Fuzzing

Esta secção apresenta, resumidamente, a base do trabalho realizado, no qual é usado fuzzing para detetar vulnerabilidades em produtos de software.

Fuzzing é uma técnica utilizada para encontrar bugs no software a ser testado. Esta foca-se na geração de inputs, seguido pela execução destes e monitorizão de como o programa se comporta com esses inputs.

Existem três tipos de técnicas de fuzzing:

- **Blackbox fuzzing**: Este tipo de fuzzers trata o programa a ser testado como uma caixa negra, não tendo qualquer tipo de informação sobre a estrutura e código fonte do programa. Isto leva a que este tipo de fuzzers tenham dificuldades a conseguir passar guardas que requerem determinados inputs, fazendo que estes tipicamente tenham uma má cobertura de código.
- **Whitebox fuzzing**: Por outro lado, temos o whitebox fuzzing. Este tipo de fuzzers usam técnicas de análise de programas para perceber que tipos de inputs conseguem chegar a certas partes do código. Isto, em conjunto com um coletor de guardas e um solucionador destas, leva a que estes tipos de fuzzers tenham melhor cobertura de código que os Blackbox fuzzing. No entanto, fuzzers deste género demoram muito tempo a gerar inputs devido ao tempo que gastam na análise do programa e a resolver as guardas.
- **Greybox fuzzing**: Fuzzers Greybox são uma mistura entre Blackbox fuzzers e Whitebox fuzzers, onde o fuzzer consegue ter uma ideia sobre a estrutura do programa usando instrumentalização. Geralmente é gerado um Id para cada basic block do programa a ser testado e durante a execução do programa é possível obter informação sobre o código executado pelo programa. Este processo é mais rápido do que a análise realizada pelos whitebox fuzzers, mas obtém-se menos informação.

AFL, ou *American Fuzzy Lop*, [10] é atualmente o Greybox fuzzer mais popular para testar programas em C/C++. É altamente eficiente, podendo executar centenas de inputs por segundo, cobrindo uma grande área da superfície de ataque do programa em, relativamente, pouco tempo. Este fuzzer funciona mantendo uma Queue de inputs que exercitarão diferentes caminhos no código [11]. O AFL recebe inputs válidos do programa, isto é, inputs que consigam correr no programa a ser testado e procede a mutar esses inputs de modo a adquirir novos casos de teste que consigam gerar novos caminhos no programa.

O AFL, sendo um greybox fuzzer, utiliza instrumentalização de código que é executada no tempo de compilação. Esta instrumentalização funciona injetando pedaços de código na aplicação em certos lugares chave. Esses pedaços de código vão ser injetados antes de cada *basic block* da aplicação. Um basic block é um conjunto de linhas de código que é executado sequencialmente, e este conjunto de linhas funciona como o Id do basic block. O AFL gera estes Ids aleatoriamente. Depois de ter gerado os Ids para cada basic block do programa, o AFL adiciona uma componente de memória partilhada para poder informar o fuzzer sobre o caminho percorrido durante a execução de um dado input. Esta informação é

partilhada em forma de cobertura por aresta. Os Ids adicionados à memória partilhada são a combinação do último Id passado e do Id corrente, gerados pela função $Prev\_Id \oplus (Cur\_Id >> 1)$. A operação *shift* é feita no Id corrente para permitir haver distinção entre a aresta resultante de A para B e de B para A. Assim, cada vez que um dado Id é processado durante a execução do programa é adicionado a aresta resultante da operação. Isto permite ao fuzzer identificar se um dado input obteve nova cobertura de código, verificando apenas a informação dos basic blocks que foram processados no final da execução.

3 Arquitetura da Solução Proposta

Esta secção apresenta a visão geral da solução proposta, bem como as componentes que a compõem.

3.1 Visão Geral

O objetivo da solução proposta é a deteção de vulnerabilidades em variantes de um mesmo programa através da reutilização de resultados de testes anteriores. Assim sendo, o propósito da solução é tentar aprender qual a melhor forma de testar um programa baseado na informação descoberta de testes anteriormente efetuados numa variante do programa.

Neste sentido, a solução foca-se em efetuar testes que permitam obter informação sobre outras variantes do mesmo programa, obtendo informação sobre a possível estrutura de programas ainda não testados.A solução para poder aprender com os testes anteriores tem de lidar com os seguintes desafios:

(1) Compreender melhor a estrutura do programa; (2) Evitar repetir testes entre variantes de programas; (3) Conseguir aprender entre sessões de testes; (4) Conseguir direcionar o fuzzer para pedaços de código interessantes em diversas variantes do programa.

A arquitetura da solução é composta por duas componentes. A componente de Instrumentalização, que nos permite obter informação sobre a estrutura do programa, e a componente de fuzzing, que executa o processo de deteção de vulnerabilidades com base na informação obtida da fase de instrumentalização.

3.2 Instrumentalização

A instrumentalização do programa a ser testado tem como intenção facilitar o processo de descoberta das vulnerabilidades existentes no programa. Irá assim permitir ao fuzzer obter informação sobre a estrutura interna do programa e a deteção de funcionalidades partilhadas entre programas. A arquitetura do processo de instrumentalização é demonstrada na Figura 1

De modo a conseguirmos obter mais informação sobre a estrutura do programa, a geração do Id de cada basic block do programa depende do conteúdo do basic block em si, ao contrário do AFL que o gera aleatoriamente. Desta forma,

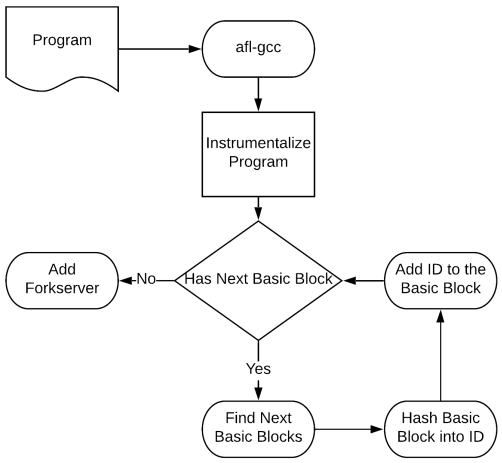

Figura 1. Arquitetura proposta para a componente de Instrumentalização

o Id de dois basic block, independentemente em que programa estejam, vai ser o mesmo se o conteúdo dos dois basic Blocks for o mesmo.

Isto permite resolver o primeiro desafio, obtendo-se assim mais informação sobre o conteúdo e a estrutura do programa, bem como o quarto desafio, focando os testes nas localizações do programa em que o código varia entre variantes do programa a testar, ou seja, as zonas onde as vulnerabilidades podem estar escondidas. O problema da deteção de funcionalidades diferentes entre programas pode ser também reduzido ao problema de deteção de basic block não partilhados entre dois programas, uma vez que qualquer bloco partilhado é um bloco em que a sua funcionalidade é também partilhada entre as duas variantes.

3.3 Fuzzing

O componente de Fuzzing permite detetar as faltas existentes no programa, usando a informação sobre a estrutura e o conteúdo do programa que foi obtida na fase anterior. Essencialmente, a fase de fuzzing foca-se em obter a maior cobertura do código da aplicação com base na geração de inputs que possam explorar o programa e que evitam cobrir áreas do código que já tenham sido testadas entre variantes.

Resolvemos o segundo desafio mencionado anteriormente mantendo em memória as arestas já vistas, independentemente da variante em que se encontram. Duas arestas vão ser iguais somente se ambos os basic blocks que as constituem são iguais. Neste sentido, caso tenhamos duas arestas iguais em duas variantes de um programa, dois basic blocks que são partilhados entre os programas, ou seja, a aresta é informação que já foi vista anteriormente. Assim, tentamos evi-

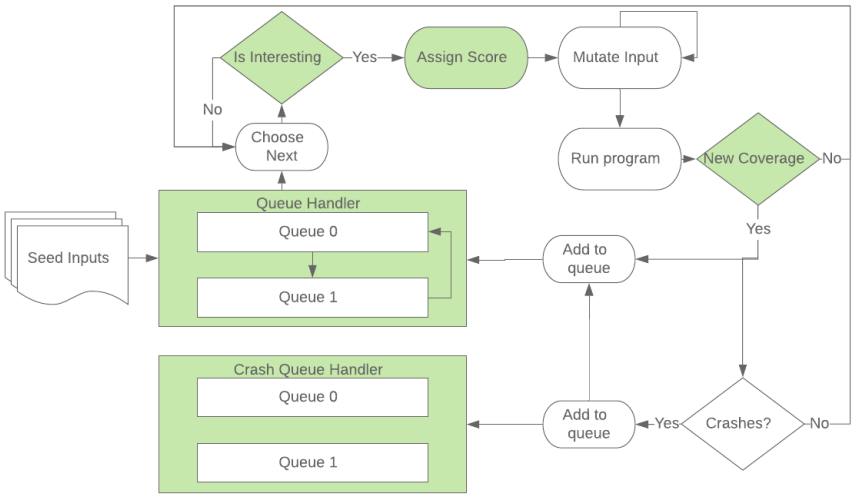

Figura 2. Arquitetura proposta para a componente de fuzzing

tar repetir testes com arestas iguais priorizando os inputs que conseguem gerar arestas nunca antes vistas em qualquer um dos programas.

Se apenas fizermos fuzzing de duas variantes de um programa em separado não aprendemos nada entre estas duas sessões de testes, perdendo informação que poderia ser importante para testar melhor qualquer uma dessas variantes. Deste modo, implementamos um mecanismo que permite aprender entre sessões de fuzzing. Este mecanismo verifica a cada intervalo de tempo se existe informação nova por aprender da última sessão de fuzzing. Esta informação é constituída por inputs que conseguem gerar novos casos de teste que permitem nova cobertura de código. Esses novos casos de teste são executados no programa e verificado se são considerados interessantes para uma dada variante do programa. Os casos de teste interessantes são adicionados à queue para mais tarde serem mutados e para poderem gerar novos inputs. O mecanismo proposto permite poupar tempo na mutação de inputs uma vez que estes são gerados anteriormente, ganhando-se tempo para explorar outros caminhos no programa. Este aspeto permite-nos resolver o terceiro desafio.

Em seguida apresentamos os módulos que pertencem ao processo de fuzzing, que estão ilustradas na Figura 2.

1. **Queue Handler**: A Queue Handler mantém e interage com todas as queues que representam o estado de exploração dos programas em teste. Cada programa tem uma queue que representa os estados de exploração do programa, contendo em si os inputs que geraram nova cobertura do programa e que poderão ainda gerar nova cobertura.
2. **Seleção do próximo caso de teste**: Este módulo escolhe o input de teste para fazer as mutações. Funciona como o AFL quando está a selecionar o teste, verificando apenas se o input de teste é considerado interessante.

3. **É interessante**: Dá prioridade a testar inputs que são mais prováveis de descobrir novos caminhos de execução, ou seja, de aumentar a cobertura da aplicação. Um input de teste é mais ou menos provável dependendo da pontuação que o input que a gerou obteve. A lógica é que inputs que conseguiram obter um resultado favorável são mais prováveis de gerar inputs que obtenham resultados favoráveis.
4. **Atribuição de pontuação**: A cada input de teste é associado uma pontuação. Essa pontuação depende do tempo de execução do input de teste, quão profundo no programa consegue ir e se descobriu nova cobertura de código. Dependendo da pontuação, um input de teste é mutado por mais ou por menos tempo. Um input de teste em que a mutação dura mais tempo tem maior probabilidade de gerar um input que consegue cobrir mais código.
5. **Mutação do Input**: O processo de mutação permite ao fuzzer explorar o programa a ser testado. Deste modo, quanto melhor for o mecanismo de mutação melhor é a exploração do programa.
6. **Crash Queue Handler**: Devido ao facto de existirem múltiplos programas a correr ao mesmo tempo, é preciso ter a capacidade de identificar que inputs causam o crash do programa. Deste modo, temos um crash queue que permite dividir os inputs que crasham o programa em si.

Por exemplo, uma tipica sessão de fuzzing funcionaria da seguinte forma:

1. Instrumentalização do programa de modo a obter informação da estrutura do programa durante a execução de um dado caso de teste.
2. Verificar se um dado tempo já passou. Se sim, passar toda a informação que foi obtida do processo de fuzzing atual para a próxima sessão de fuzzing e começar um novo processo de fuzzing na variante do programa, segundo o processo descrito para a solução do desafio três.
3. Escolher um dado caso de teste. A seleção é efetuada mantendo um índice que passa por todos os testes na Queue. Uma vez que passemos por todos os testes presentes na queue, o índice aponta de novo para o caso inicial.
4. Verificar se esta escolha é interessante de acordo com o que foi definido na componente que define se um dado input é interessante e que não repetimos nenhum teste. Caso não seja interessante voltamos para o ponto 2.
5. Cálculo da pontuação do input de teste, tendo em conta a quantidade de tempo que demorou a correr no programa e se gerou nova cobertura de código. Essa cobertura verifica também se obteve nova informação que não é partilhada entre as variantes do programa.
6. Mutação do input tendo em conta a sua pontuação. Quanto maior for a pontuação maior o tempo que um dado input é mutado. Essas mutações são corridas no programa, caso descubram nova cobertura no programa são guardas no Queue Handler para repetir o processo novamente.
7. Voltar ao passo 2.

4 Implementação do PandoraFuzzer

O protótipo apresentado neste paper é baseado no greybox fuzzer AFL. A versão atual da ferramenta permite a deteção de qualquer tipo de vulnerabilidades que

o AFL deteta. A ferramenta foi escrita em C. É composta por uma componente de instrumentalização de injeção e de fuzzing. Nesta secção descrevemos como foi implementada cada componente.

4.1 Instrumentalização

Esta componente foi maioritariamente implementada em C. Sendo que a única subcomponente que é escrita em Assembly é o código que é injetado no programa a ser testado. A componente desenvolvida funciona em substituição do gcc ou do clang, permitindo injetar o código assembly no programa a ser testado.

A geração do Id do basic block começa por dividir o código de cada basic block em diversas linhas, removendo qualquer linha de código que não afete a execução do mesmo. Em seguida, são removidos os registos, por o seu conteúdo ser muito volátil, e os comandos restantes são concatenados. Por fim, é utilizado a função de hashing de Pearson de 16 bits sobre a concatenação resultante para gerar o Id do bloco. O uso do hash garante uma melhor distribuição dos Ids, isto é, haver menos basic blocks distintos com o mesmo Id.

É usado a cobertura por arestas para obter informação sobre a estrutura interna do programa, significando que os Ids obtidos da execução de um dado input são a combinação de dois Ids em sequência e possibilita obter informação sobre a interação dos basic blocks.

4.2 Fuzzing

Esta componente foi desenvolvida toda em C. A lógica de fuzzing segue a mesma que o AFL. Em seguida é apresentando como foram implementados os três módulos que foram inseridos no AFL.

- **Queue Handler**: O Queue Handler foi implementado usando uma *Linked List* onde cada valor representa um caso de teste a ser testado.
- **Mutação de Input**: A mutação do input é dividida em duas fases. A fase Determinística, onde as mutações seguem todas valores já pré-definidos. E a fase Havoc, onde as mutações são aleatórias. Dentro das mutações Determinísticas temos vários tipos de mutações, trocar o valor do bit onde passamos por cada bit iterativamente e trocamos o valor, trocar o valor do Byte que é feito de maneira semelhante à troca do bit, adições aritméticas onde simplesmente somamos ou subtraímos certos valores ao input e usamos valores interessantes que são valores que são prováveis gerar faltas, como por exemplo o máximo valor que um *integer* pode suportar.
- **Crash Queue Handler**: A mesma implementação que foi aplicado no Queue Handler foi implementada no Crash Queue Handler onde cada valor representa um input que permite crashar o programa.

5 Avaliação

O objetivo da avaliação experimental é testar a eficácia do protótipo implementado na deteção de vulnerabilidades em programas. As experiências foram

efetuadas tendo em consideração algumas diretrizes para testar e avaliar fuzzers [9]. Antes de realizar as experiências, foi realizado uma auditoria manual às aplicações para detetar o número de linhas de código e o número de ficheiros de cada programa para termos uma visão do tamanho dos programas a testar.

A ferramenta foi avaliada com base em duas características principais: o número de vulnerabilidades detetadas e a cobertura de código que proporciona ao programa. A prestação da ferramenta foi avaliada com quatro programas disponíveis no pacote *binutils* do sistema operativo Linux. A Tabela 1 apresenta toda a informação importante sobre esses programas, onde apresentamos o resultado da auditoria realizada manualmente, e o número máximo de vulnerabilidades e de caminhos descobertos de todos os processos de fuzzing. É importante mencionar que os resultados apresentados são a média dos resultados de 10 sessões de fuzzing por programa, durante 1 hora cada. Na tabela é também apresentado o número máximo de crashes e de caminhos descobertos no procedimento de fuzzing. O input inicial proporcionado para quase todos os programas foi apenas uma única string vazia. A única aplicação onde os testes gerados foram mais complexos é na aplicação *readelf*. Isto foi devido ao baixo número de caminhos detetados no processo de fuzzing por qualquer uma das ferramentas.

Programas	LoC	Número de ficheiros	Número de crashes	Número de paths
cxxfilt	5994	17	406	2739
readelf	13275	3	0	644
strings	5899	16	0	81
size	5807	14	18	758

Tabela 1. Informação sobre todos os programas testados

5.1 Deteção de Vulnerabilidades

A primeira fase da avaliação consiste na avaliação da performance da ferramenta em encontrar vulnerabilidades e compará-las com o AFL.

A Tabela 2 mostra os resultados da avaliação, onde observa-se que na aplicação *cxxfilt* a ferramenta detetou uma quantidade significativamente maior de vulnerabilidades que o AFL. No entanto, no programa *size* o AFL detetou mais uma vulnerabilidade que o PandoraFuzzer.

Programas	AFL	PandoraFuzzer
cxxfilt	60	74
readelf	0	0
strings	0	0
size	8	7

Tabela 2. Binutils Vulnerabilidades descobertas

5.2 Cobertura de Código

Na fase seguinte, foi comparado a cobertura de código proporcionada pelo protótipo implementado contra a cobertura oferecida pelo AFL. Esta informação

também é importante porque, teoricamente, quanto mais cobertura de um programa um fuzzer proporciona a uma aplicação mais vulnerabilidades poderá detetar no mesmo espaço de tempo. A lógica é que como estamos a cobrir mais código do programa, mais hipóteses temos de descobrir as vulnerabilidades que poderão estar escondidas.

Os resultados apresentados na Tabela 3 mostram que a ferramenta, em média, cobre mais código de um programa do que o AFL. Conseguindo, em alguns dos casos, uma maior cobertura que o AFL.

Programas	AFL	PandoraFuzzer
cxxfilt	2261	2349
readelf	305	644
strings	65	74
size	609	583

Tabela 3. Binutils cobertura de codigo por caminhos

5.3 Discussão dos resultados

O uso de informação sobre a estrutura do programa obtida durante um processo de fuzzing realizado anteriormente permitiu obter mais informação sobre a variante do programa. Permitindo ao PandoraFuzzer obter mais informação do que o AFL no mesmo intervalo de tempo. As únicas divergências entre o processo de fuzzing em qualquer programa é o uso de informação obtida num processo de fuzzing anterior e a informação sobre os caminhos que já foram testados em testes anteriores. No caso do *cxxfilt*, foi possível obter mais vulnerabilidades no mesmo tempo devido ao facto de evitarmos caminhos já percorridos, obtendo novas vulnerabilidades. No entanto, isto não aconteceu em todos os casos. Nos resultados experimentais houve um único programa no qual o AFL obteve mais informação sobre a estrutura interna do programa e um maior número de vulnerabilidades.

Após uma análise das vulnerabilidades geradas nesse período de fuzzing, como a aplicação *size* em si, foi possível obter mais informação para perceber a razão porque isto aconteceu. Em primeiro lugar, é importante mencionar que as vulnerabilidades obtidas nessa aplicação eram semelhantes. Ou seja, ambas as ferramentas exercitaram os mesmos caminhos. Podemos concluir que ambos os fuzzers exploraram a mesma área de código para conseguirem obter essa informação. Mas, o PandoraFuzzer exerceu um maior esforço para explorar as áreas que foram modificadas entre cada variante. Após uma análise de cada variante, foi descoberto que as vulnerabilidades presentes no *size* estavam presentes não nessas variantes, mas sim no código que se tinha mantinho constante. Como o AFL não foca os testes nas áreas modificadas entre variantes, este detetou mais informação sobre as vulnerabilidades que se mantiveram entre variantes.

6 Trabalho Relacionado

Nesta secção é apresentado uma seleção de trabalhos relacionados com o desenvolvimento progressivo de fuzzers. Conectado com as diferentes componentes dos fuzzers em si.

Em [2] foi introduzido um Greybox direcionado que foi implementado na ferramenta, AFLGo. Este fuzzer gera inputs com o objetivo de estes atingirem locais específicos do código. Os autores desenvolveram uma heurística para minimizar a distância de um dado input de teste para o local onde querem chegar. As suas experiências demonstraram que o fuzzing direcionado tem melhores resultados do que análise simbólica, tanto em termos de eficiência como em termos de eficácia.

O trabalho desenvolvido por Grieco et al. [7] foca-se na previsão de quanto provável um input de teste pode encontrar uma vulnerabilidade no software em teste. Isto é feito focando-se no uso de análise estatística, usando técnicas de machine learning que analisa o código binário dos programas. Os resultados da avaliação demonstram que analisar uma pequena percentagem do programa marcado como tendo uma potencial vulnerabilidade, aumenta significativamente a velocidade do processo de fuzzing.

Em [3] foi desenvolvido um greybox fuzzer que combina a análise estática com o fuzzing direcionado. Os resultados experimentais obtidos mostram que a ferramenta tem boa capacidade para conseguir chegar aos locais alvo.

Angora [4] é um fuzzer baseado em mutações que aumenta o número de caminhos de execução encontrados resolvendo as guardas que encontra durante a execução do programa sem ter de usar execução simbólica. Na fase de avaliação, Angora superou o AFL em cobertura de linhas, cobertura de arestas e encontrou mais vulnerabilidades.

O LibFuzzer [1] funciona especificando um ponto de entrada no programa que queremos testar, uma função que aceita certos dados e faz algo de interessante com eles. Com esta função podemos direcionar o input para a função que queríamos. Como o AFL, LibFuzzer gera inputs baseados no input inicial.

Outro fuzzer interessante é honggfuzz [5], um fuzzer evolucionário orientando para segurança capaz de gerar uma grande quantidade de testes por segundo. Este fuzzer já foi usado para detetar uma grande quantidade de problemas de segurança apontados em [6].

Em [8] utilizam analise estática e analise dinâmica do programa para gerar inputs que consigam gerar inputs interessantes para a aplicação que consigam cobrir nova cobertura do código a ser testado. Obtém informação à priori da aplicação extraindo informação sobre o fluxo de dados, como por exemplo utilizando *taint analysis*, para inferir sobre a estrutura do programa. Implementaram uma ferramenta, VUzzer, e concluíram que análise prévia da estrutura do programa é uma estratégia viável e escalável para melhor os resultados de fuzzing.

Apesar de muitas das ferramentas mencionadas acima serem direcionadas, nenhuma delas utiliza informação obtida durante a execução de testes posteriores numa variante do programa para melhor analisar a seguinte variante.

7 Conclusão

Este artigo apresentou uma arquitetura para a deteção automática de vulnerabilidades em variantes de aplicações com funcionalidades partilhadas, tendo como base o AFL ou American Fuzzy Lop. A ferramenta aqui desenvolvida monitoriza as aplicações em execução enquanto obtém informação sobre a estrutura interna do programa, de modo a poder direcionar os testes para áreas no código consideradas interessantes. A arquitetura foi implementada e avaliada, mostrando a sua eficácia no que toca na deteção de vulnerabilidades e cobertura de código.

Agradecimentos: Este trabalho foi parcialmente suportado pelos fundos nacionais através da Fundação para a Ciência e a Tecnologia (FCT) com referência ao projeto SEAL (FCT-029058) e à Unidade de Investigação LASIGE (UID/CEC/00408/2019).

Referências

1. libfuzzer. https://llvm.org/docs/LibFuzzer.html/. [Accessed in 20/10/18], 2018.
2. M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344, New York, NY, USA, 2017. ACM.
3. H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2095–2108, New York, NY, USA, 2018. ACM.
4. P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. volume abs/1803.01307, pages 711–725, 2018.
5. Google. honggfuzz. https://github.com/google/honggfuzz/. [Accessed in 28/10/18], 2018.
6. Google. honggfuzz Trophies. http://honggfuzz.com/. [Accessed in 28/10/18], 2018.
7. G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 85–96, New York, NY, USA, 2016. ACM.
8. I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 49–64, Washington, D.C., 2013. USENIX.
9. G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. ACM.
10. M. Zawlewski. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/. [Accessed in 01/11/18], 2017.
11. M. Zawlewski. AFL Technical Details, 2018. http://lcamtuf.coredump.cx/afl/technical_details.txt/.[Accessed in 30/10/18].