# Isolamento de Falhas em Redes Definidas por Software

João Miranda[1], Nuno Machado[2], and Luís Rodrigues[1]

[1] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa,
{joaoshmiranda,ler}@tecnico.ulisboa.pt
[2] HASLab, INESC TEC & Universidade do Minho, nuno.a.machado@inesctec.pt

**Resumo** As redes definidas por software (do Inglês, *Software Defined Networks*, SDNs) têm vindo a afirmar-se como uma das mais promissoras abordagens para simplificar a configuração e gestão de equipamentos. No entanto, as SDNs não são imunes a erros tais como ciclos no encaminhamento, buracos negros, encaminhamento sub-óptimo, entre outros. Estes erros são tipicamente causados por falhas na especificação ou por *bugs* nos equipamentos. Se as primeiras podem ser, em grande parte, eliminadas através da utilização de ferramentas que fazem a validação automática de uma especificação antes da sua instalação, os *bugs* (e/ou avarias) nos equipamentos (muitas vezes de natureza não determinista) geralmente só conseguem ser detectados em tempo de execução. Este artigo propõe uma nova técnica para facilitar o isolamento de falhas nos equipamentos em redes SDN. Esta técnica combina a utilização de ferramentas de validação formal (para obter os caminhos esperados para os pacotes) e ferramentas de registo de pacotes (para obter os caminhos observados) para realizar uma análise diferencial que permite identificar com exactidão qual o equipamento onde ocorreu a falha.

## 1 Introdução

As redes definidas por software (do Inglês, *Software Defined Networks*, SDNs) emergiram como um paradigma promissor para gerir as redes do futuro. Este paradigma permite separar o problema de encaminhamento de pacotes em dois planos distintos: o *plano de dados* (encarregue de encaminhar os pacotes entre dispositivos) e o *plano de controlo* (que define qual o caminho que os pacotes devem seguir). Para além disso, permite que os equipamentos que materializam o plano de dados sejam configurados remotamente a partir de um ponto de controlo logicamente centralizado, usando para isso interfaces normalizadas como o OpenFlow [11]. Com estas vantagens, as SDNs apresentam-se com o potencial para simplificar significativamente a gestão de redes [10].

Apesar destes avanços, a depuração de erros em redes definidas por software continua a ser uma tarefa complexa e morosa. Se, por um lado, a existência de um controlador logicamente centralizado, programado por software, permite reutilizar técnicas de teste e verificação usadas noutros domínios da informática, por outro, as avarias nos equipamentos, juntamente com os *bugs* no software que

estes executam, continuam a gerar erros que só podem ser detectados em tempo de execução. Considerando que as redes possuem um grande número de equipamentos, por vezes na ordem dos milhares, torna-se fundamental desenvolver ferramentas que ajudem os gestores de rede nas tarefas de depuração.

Na literatura recente, têm sido propostas várias ferramentas que permitem evitar a instalação de configurações incoerentes ou erradas [2,7,9,12,13,5]. Por exemplo, algumas ferramentas usam técnicas de verificação de modelos (do Inglês, *model checking*) para assegurar que requisitos invariantes não são violados, quer offline [2,9], quer no momento da instalação [7]. Uma vez que estas técnicas só permitem validar a especificação do programa, é necessário usar mecanismos complementares para lidar com falhas que ocorrem em tempo de execução.

Por sua vez, ferramentas de depuração como o OFRewind [13] e o ndb [5] recorrem a instrumentação, registo de eventos e mecanismos de reprodução de execuções. Infelizmente, a tarefa de inspeccionar um histórico de eventos para isolar os componentes com falha ainda se pode revelar bastante morosa [3,12].

Neste artigo propomos o NetSheriff, um sistema que permite automatizar a identificação do dispositivo responsável por uma falha numa rede SDN. O NetSheriff combina características das ferramentas de validação e das ferramentas de depuração para facilitar a identificação da causa das falhas. Para cada classe de equivalência de encaminhamento, o NetSheriff usa o modelo da rede e os comandos fornecidos pelo controlador para gerar os caminhos esperados para as diversas classes de equivalência. Em tempo de execução, o NetSheriff regista quais os caminhos observados. Por fim, através de uma análise comparativa entre o caminho esperado e o caminho observado, o NetSheriff identifica o equipamento que se desviou do comportamento correcto.

Avaliámos o NetSheriff usando diferentes topologias de rede e diferentes tipos de falha. Os resultados obtidos mostram que a ferramenta consegue identificar correctamente o equipamento responsável pelo erro de forma precisa.

O resto do artigo está estruturado da seguinte maneira. A Secção 2 apresenta uma breve revisão do trabalho relacionado. A Secção 3 descreve a arquitectura do NetSheriff. A Secção 4 descreve a implementação do sistema e a Secção 5 apresenta a sua avaliação. A Secção 6 resume os pontos-chave do artigo.

## 2 Trabalho Relacionado

Durante os últimos anos, têm vindo a ser propostas diversas técnicas que permitem aumentar a fiabilidade das redes definidas por software. Nesta secção, apresentamos uma panorâmica das principais contribuições para o teste e verificação de SDNs, bem como para o isolamento das causas de erros neste tipo de redes.

*Teste e Verificação.* O NICE [2] é um sistema que combina verificação de modelos e execução simbólica para descobrir, de forma automática, erros em SDNs. O NICE modela o comportamento da rede como um grafo de transições entre estados; para cada um destes estados, o NICE verifica se uma determinada propriedade, que se pretende assegurar, é violada. Em caso de violação do requisito

invariante, o NICE usa execução simbólica para gerar uma sequência de pacotes que despoleta o erro.

De forma semelhante, o NetPlumber [6] e o Veriflow [7] usam verificação de modelos para validar *a priori* as configurações que o controlador pretende instalar. Para tal, estas ferramentas interceptam os comandos enviados para os dispositivos e verificam que estes não violam os requisitos invariantes desejados.

Por sua vez, o VeriCon [1] verifica especificações de SDNs em tempo de compilação, validando a sua correcção para as topologias e sequências de pacotes admissíveis. O NetSheriff não pretende competir com as ferramentas acima referidas, mas sim complementá-las, focando-se em detectar erros nos dispositivos (e não nas especificações).

*Isolamento da Causa do Erro.* O NetSight [4] e o OFRewind [13] são ferramentas que pretendem facilitar a depuração de redes SDN. O NetSight regista históricos de pacotes e oferece uma ferramenta de depuração (denominada *ndb*) que facilita a navegação nestes históricos. O OFRewind efectua o registo de históricos e de pacotes para, posteriormente, permitir a reprodução de uma execução.

Mesmo com recurso a estas ferramentas, o trabalho de localizar a causa nestes casos continua a obrigar a um esforço significativo, visto que um histórico pode conter um número elevado de eventos. O NetSheriff pretende atacar este problema, ajudando a identificar os eventos do histórico que são relevantes para a falha e quais os dispositivos associados ao erro.

## 3 NetSheriff

Nesta secção descrevemos o NetSheriff. Em primeiro lugar, apresentamos uma visão geral da arquitectura do sistema. De seguida, descrevemos como o NetSheriff usa ferramentas de verificação para calcular os caminhos esperados dos pacotes e quais os mecanismos que utiliza para capturar os caminhos observados. Finalmente, explicamos como esta informação é usada para localizar a falha.

### 3.1 Arquitectura

O NetSheriff é um sistema que permite detectar automaticamente caminhos de tráfego incorrectos e identificar o dispositivo responsável pelo erro. O NetSheriff é composto por quatro componentes, tal como ilustrado na Figura 1: o *planificador*, o *instrumentador*, *o coleccionador* e o *comparador*.

*Planificador.* O planificador é responsável por calcular os caminhos esperados para cada classe de equivalência de encaminhamento. Estes caminhos são modelados por *grafos de propagação* [7], que são uma representação dos percursos que os pacotes de uma dada classe de equivalência devem percorrer na rede. Os vértices deste grafo representam os nós da rede e as arestas representam os elos de ligação entre os nós. Por exemplo, seja $\mathcal{F}$ o grafo de propagação para uma

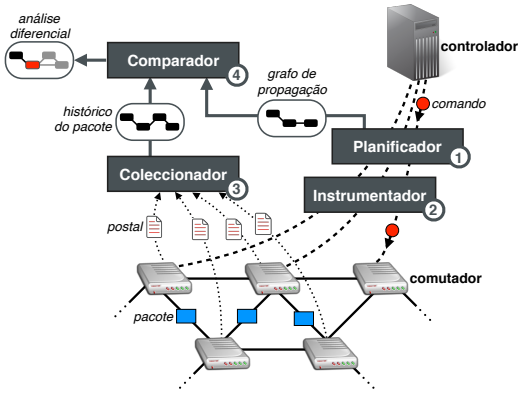

Figura 1: Visão geral do NetSheriff.

classe de equivalência $CE$. A aresta $A \rightarrow B$ em $\mathcal{F}$ indica que o equipamento $A$ encaminha todos os pacotes da classe $CE$ para o equipamento $B$.

Concretamente, o planificador intercepta todos os comandos do controlador, calcula as classes de equivalência afectadas por esses eventos e gera os respectivos grafos de propagação. Estes grafos são depois enviados para o comparador, para análise posterior. A obtenção dos grafos em tempo de execução, de forma expedita e a permitir a sua análise num horizonte temporal curto, pode ser um desafio. Na Secção 3.2 discutimos como o NetSheriff aborda este problema.

*Instrumentador.* O instrumentador tem como objectivo possibilitar a gravação da informação necessária para reconstruir os caminhos percorridos pelos pacotes na rede. Para este efeito, o instrumentador aproveita-se da arquitectura das SDN, interceptando mensagens entre o controlador e os comutadores (de notar que tanto o planificador como o instrumentador são transparentes para os comutadores e para o controlador).

Em particular, o instrumentador estende cada regra enviada pelo controlador, acrescentando-lhe uma acção para duplicar os pacotes por ela afectado. Os pacotes duplicados (designados por *postais*) são enviados com destino ao coleccionador. No entanto, antes disso, são acrescentadas acções para modificar estes postais de forma a que seja registada toda a informação que permite reconstruir o histórico de um pacote (*e.g.,* o identificador do comutador, portos de saída e modificações aos cabeçalhos).

*Coleccionador.* O coleccionador consiste num servidor (central ou distribuído) que recebe os postais enviados pelos comutadores e organiza-os com o propósito de gerar múltiplas colecções distintas, denominadas históricos de pacotes. Cada histórico de pacote corresponde ao conjunto de todos os postais gerados pela travessia de um determinado pacote na rede. O histórico de um pacote permite assim construir o caminho percorrido por esse pacote e saber as alterações realizadas ao seu cabeçalho em cada ponto.

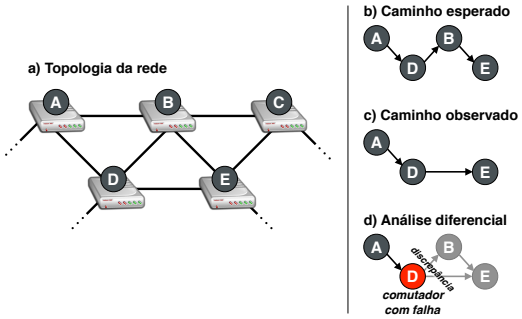

Figura 2: Ilustração da análise diferencial executada pelo comparador.

À semelhança dos grafos de propagação, os históricos de pacotes também são enviados para o comparador, para sua análise futura. Contudo, os desafios que se colocam aqui são o de processar eficientemente um grande número de postais e de lidar com a recepção não ordenada de postais. A Secção 3.3 discute estes aspectos, bem como o modo como é feito o encaminhamento dos postais.

*Comparador.* O comparador é o componente encarregue de fazer a análise diferencial entre os grafos de propagação e os históricos de pacotes. Esta análise é feita projectando os históricos nos grafos e detectando discrepâncias. O caso mais simples acontece quando o pacote segue um caminho único entre um ponto de origem e um ponto de destino, sendo que tanto o grafo de propagação como o histórico de pacotes representam uma sequência linear de comutadores. Neste caso, basta verificar se ambos os caminhos correspondem à mesma sequência de comutadores e, caso tal não aconteça, em que comutador se dá o desvio. Isto é ilustrado na Figura 2, que representa uma rede SDN com 5 comutadores: $A, B, C, D$ e $E$. Considere-se uma classe de equivalência de encaminhamento entre $A$ e $E$ que deve seguir o seguinte caminho $A \rightarrow B \rightarrow C \rightarrow D$ (Figura 2b). Devido a uma falha, o caminho observado é $A \rightarrow D \rightarrow E$ (Figura 2c). Ao fazer a análise diferencial, o comparador verifica que existe uma disparidade no comutador $D$. Assim, o NetSheriff consegue identificar este comutador como a fonte do erro.

Nos casos mais complexos podem existir *multicasts* ou *broadcasts*, o que significa que, em determinados pontos, os caminhos podem dividir-se em múltiplas ramificações. Além disso, podem existir alterações nos cabeçalhos (o que até é provável, para garantir que os pacotes são correctamente recebidos pelas várias máquinas), que são importantes ter em conta aquando da análise diferencial. Caso contrário, pode obter-se caminhos sem ramificações, o que tornaria inviável a distinção entre vários tipos de erros, tais como os apresentados na Secção 5. O NetSheriff lida com este problema fazendo uma combinação dos diversos grafos possíveis para um dado histórico de pacotes, tendo em conta em que zonas pode ser feita a combinação (*i.e.,* verificando onde ocorrem as modificações dos cabeçalhos). O grafo é depois projectado contra os postais dos históricos corres-

pondentes tendo em conta as modificações observadas e, caso se detecte alguma divergência, o erro pode ser classificado de acordo com características do grafo analisado. O algoritmo de comparação de grafos é descrito em pormenor na Secção 4.

### 3.2 Cálculo dos Grafos de Propagação

Tal como referido, o NetSheriff baseia-se na utilização de grafos de propagação, que capturam os caminhos que os pacotes devem seguir na rede. Estes percursos podem ser alterados sempre que uma nova configuração é instalada. Por razões de eficiência, o NetSheriff só recalcula os grafos de propagação para as classes de equivalência que são afectados pelas regras que são alteradas pelo controlador. O NetSheriff utiliza o Veriflow [7], beneficiando da capacidade deste sistema para calcular os grafos de propagação.

Mais concretamente, o planificador mantém uma *trie* (*i.e.,* uma árvore de prefixos multidimensional) que associa as regras instaladas nos comutadores com as classes de equivalência que estas afectam. Nesta estrutura, cada nível da árvore representa um bit de uma regra de encaminhamento (correspondente a um bit do cabeçalho de um pacote). Cada nó da árvore pode ter um de três valores: '0', '1', e '*' (em que '*' representa um valor polivalente). Desta forma, cada nó da árvore pode gerar três ramificações, uma por cada um destes valores. As folhas da árvore mantêm tuplos com a forma $\langle c, r \rangle$ em que $c$ é o identificador de um comutador e $r$ uma regra de encaminhamento que é aplicada aos pacotes correspondentes ao cabeçalho indicado pelo caminho da raíz até essa folha. Cada dimensão da árvore (ou sub-árvore) representa um campo do cabeçalho de um pacote. Sempre que o controlador decide alterar uma regra de encaminhamento, o planificador usa as árvores para identificar as classes de equivalência afectadas por uma regra.

Por exemplo, considere-se uma configuração com uma regra para o prefixo 10.1.0.0/16, à qual se acrescentava uma regra de maior prioridade para o prefixo 10.1.1.0/24. Isto criaria uma nova classe de equivalência que é um sub-conjunto da classe de equivalência original, obrigando a definir novos intervalos, nomeadamente [10.1.0.0, 10.1.0.255], [10.1.1.0, 10.1.1.255] e [10.1.2.0, 10.1.255.255], e a calcular um novo grafo de propagação para a classe de equivalência associada ao intervalo do meio.

### 3.3 Cálculo dos Históricos de Pacotes

O NetSheriff obtém os históricos dos pacotes recolhendo os postais que são gerados pelos dispositivos da rede. Isto obriga o coleccionador a armazenar todos os postais associados a cada pacote, o que pode ser uma quantidade de informação significativa. Para realizar esta tarefa de forma eficiente, recorremos à utilização do NetSight [4], uma ferramenta para geração, captura e processamento de postais que, para redes de grande dimensão, permite a utilização de vários servidores de recolha na rede.

*Geração e captura.* O NetSight implementa o mecanismo para duplicação de pacotes descrito na Secção 3.1. Quanto à propagação dos postais, pode ser realizada de duas formas diferentes: através da rede de produção, consumindo parte da largura de banda disponível devido ao fluxo de postais, mas evitando uso de comutadores adicionais; ou, pelo contrário, conectando comutadores adicionais que ligam os comutadores da rede a um servidor NetSight. Tanto num caso como no outro, é possível utilizar diversos servidores dispersos na rede de forma a minimizar o tráfego essencial para esta fase. Neste caso, é necessária uma fase posterior para garantir que postais gerados pelo mesmo pacote sejam armazenados no mesmo servidor.

*Processamento.* O NetSight concretiza mecanismos de coordenação entre os servidores de recolha, de forma a que estes distribuam a carga entre si. Por exemplo, o NetSight permite aplicar uma função de dispersão (*hash*) ao cabeçalho ou ao conteúdo dos pacotes para distribuir postais pelos diversos servidores, mas garantindo que postais que irão pertencer ao mesmo histórico se mantém no mesmo servidor. Cada servidor de recolha mantém uma tabela de caminhos, onde são armazenados os postais associados a cada histórico armazenado localmente. Esta tabela é materializada por uma estrutura de armazenamento do tipo chave-valor, indexada pelo identificador do pacote (*i.e.,* o resultado da função de dispersão). No caso do NetSheriff, aplicamos a função de dispersão ao conteúdo dos pacotes para garantir que os históricos contém todos os postais referentes aos respectivos pacotes, mesmo que estes sofram modificações aos seus cabeçalhos.

O NetSheriff requer que os históricos dos postais sejam ordenados para permitir que, posteriormente, o comparador possa realizar a análise diferencial. Esta ordenação é feita usando informação contida nos postais, bem como informação sobre a topologia da rede. A sequência ordenada de postais constitui assim o histórico de pacotes que é enviado para o comparador.

## 4 Concretização

Como referido nas Secções 3.2 e 3.3, adaptámos o VeriFlow para concretizar o planificador do NetSheriff, e adaptámos componentes do NetSight para concretizar o instrumentador e o coleccionador. Nomeadamente, para implementar o instrumentador utilizámos o *flow table state recorder (FTSR)* e para implementar o coleccionador usámos o servidor NetSight. O primeiro destes dois componentes opera originalmente como um *proxy* posicionado entre o controlador OpenFlow e os comutadores, tal como acontece com o VeriFlow. Como tal, ligámos ambos os *proxies* entre si, de forma a que o VeriFlow actue como um *proxy* entre o controlador e o FTSR, e este último actue como um *proxy* entre o VeriFlow e os comutadores. O servidor NetSight fica à escuta no porto de uma máquina designada para o efeito. Neste caso em concreto, num porto criado pelo emulador de redes MiniNet[8], que foi o ambiente escolhido para testar o nosso protótipo.

Apesar do comparador não ser uma extensão a nenhum destes sistemas, mas sim um componente cujos *inputs* são gerados por essas extensões, neste protótipo

o código do comparador foi acrescentado ao código do servidor NetSight para evitar mais um fluxo de comunicação e mais um programa em execução, facilitando o desenvolvimento e testes. As alterações efectuadas às versões originais do VeriFlow e do NetSight foram as seguintes.

- O VeriFlow calcula automaticamente as classes de equivalência e os respectivos grafos de propagação aquando da recepção de uma mensagem `flow_mod`. No NetSheriff, aproveitamos este facto para gerar o caminho esperado de um pacote a partir dos grafos de propagação produzido pelo VeriFlow. O caminho esperado é representado por uma sequência de ligações entre comutadores. Mais concretamente, o caminho esperado é composto por pares de identificadores ⟨*comutador origem, comutador destino*⟩. Com o objectivo de obter este caminho, alterámos o VeriFlow para enviar os grafos de propagação, associados à respectiva classe de equivalência, para o servidor NetSight. Assim, o par ⟨*classe de equivalência, grafo de propagação*⟩ é enviado através da API do servidor que foi estendida com a operação `change_path_request`.
- O servidor NetSight foi estendido para processar o par mencionado acima. Quando este par é recebido, o servidor NetSight reconstrói a classe de equivalência e o grafo de propagação (recebidos sobre a forma de texto) e instala um *filtro de históricos de pacotes* para obter os pacotes pertencentes à classe de equivalência correspondente. Dado que, por definição, os pacotes não pertencem a mais do que uma classe de equivalência, temos a garantia que os resultados do filtro são únicos. Por outras palavras, o histórico do pacote retornado pelo filtro corresponde a um só caminho esperado.

*Algoritmo de Comparação de Grafos.* Tendo disponíveis os caminhos reais e esperados, procede-se à análise diferencial para verificar se os caminhos percorridos pelos pacotes correspondem aos caminhos esperados.

O comparador reconstrói os grafos de propagação pela mesma ordem que a ordenação topológica realizada pelo NetSight. Como tal, quando recebe um histórico de pacotes, pode assumir que assim que é detectada uma diferença nos caminhos, essa diferença existe, sem ter que analisar o resto dos postais. Por esta razão, o comparador começa por efectuar uma pesquisa em largura no grafo de propagação de forma a detectar falhas o mais cedo possível no caminho. Este facto tem importância porque uma falha num comutador pode fazer com que o resto do grafo de propagação deixe de fazer sentido.

As divergências são encontradas por comparação entre os vértices do grafo de propagação e o identificador do comutador associado ao próximo postal do histórico. Antes de processar cada vértice, o comparador analisa a informação guardada nos postais para verificar se houve modificações no cabeçalho do pacote durante a sua travessia na rede. Caso existam modificações, combinam-se os grafos correspondentes às classes de equivalência dos diferentes cabeçalhos. De seguida, o comparador compara cada vértice vizinho do grafo de propagação com os próximos postais no histórico do pacote. Quando há correspondência entre o vértice vizinho e o próximo postal no histórico, a aresta do grafo fica *marcada*. Em caso de divergência, considera-se a aresta *não marcada* ou *inesperada*.

| ID | Descrição do Erro | Modo de Detecção |
|---|---|---|
| #1 | Comutador reencaminha um pacote que devia ser descartado. | Observa-se que uma ou mais arestas inesperadas partem de vértices sem arestas de saída esperadas. |
| #2 | Comutador reencaminha um pacote para todos os portos esperados e para portos adicionais não previstos na configuração. | Observa-se que uma ou mais arestas inesperadas partem de vértices com pelo menos uma aresta de saída esperada. |
| #3 | Comutador envia um pacote para porto(s) não previsto(s) na configuração, mas o pacote chega aos destinos correctos. | Observa-se caso #1 ou #2; além disso, é possível percorrer o grafo da projecção desde a origem até aos vértices finais do grafo original (*i.e.,* os que não têm arestas de saída) através de arestas esperadas marcadas e arestas inesperadas calculadas pelo NetSheriff. |
| #4 | Comutador reencaminha um pacote apenas para um subconjunto (não vazio) dos portos esperados (quando o tamanho deste conjunto é maior que 1). | Observa-se uma ou mais arestas esperadas não marcadas cujo vértice de origem tem outras arestas esperadas (e pelo menos uma delas foi marcada). |
| #5 | Comutador descarta um pacote inesperadamente. | Observa-se uma ou mais arestas esperadas não marcadas cujo vértice de origem não tem arestas de saída esperadas. |

Tabela 1: Categorias de erros suportadas pelo NetSheriff e respectiva descrição do modo como são detectadas.

A classificação das arestas é feita consoante as características do grafo observado e de acordo com categoria do erro em questão. A Tabela 1 enumera as categorias de erros consideradas neste trabalho, descrevendo também as características dos grafos analisados na presença desses mesmos erros. Na tabela, consideram-se dois tipos particulares de arestas: *aresta esperada* e *aresta inesperada*. Designa-se por aresta esperada, uma aresta que pertence ao grafo de propagação (ou seja, a aresta indica uma porção do caminho do pacote esperado na rede). Designa-se por *aresta inesperada*, uma aresta que não existia no grafo de propagação inicial, mas que é criada pela projecção entre o vértice no grafo e o comutador associado ao próximo postal do histórico. Por outras palavras, uma aresta inesperada indica que o pacote foi encaminhado para um comutador que não estava previsto, segundo a configuração da rede.

Em suma, quando o comportamento da rede está correcto e corresponde à sua configuração, todas as arestas esperadas no grafo de propagação serão marcadas pelo comparador. Além disso, não surgem arestas inesperadas no grafo da projecção. Por outro lado, na presença de erros, estas condições não se verificam simultaneamente.

É interessante referir que os postais gerados pelo NetSight transportam o número da configuração do equipamento no momento em que o pacote foi encaminhado. Isto permite ao NetSheriff associar os caminhos observados às versões correspondentes dos caminhos esperados. Esta característica permite também estender o NetSheriff para validar propriedades dos caminhos durante um processo de reconfiguração, embora não tenhamos explorado esta possibilidade no protótipo actual.

## 5 Avaliação

Avaliámos o protótipo do NetSheriff tendo como critério principal a eficácia na identificação dos comutadores em falha, bem como a capacidade de distinguir diferentes tipos de erros. Para esse efeito, foram realizadas experiências com vários tipos de erros que podem surgir nestes equipamentos. As experiências foram realizadas utilizando o MiniNet [8], versão 2.2.1, numa máquina Intel i7-720QM com 8GB RAM DDR3, 250 GB SSD e Ubuntu 14.04. A sobrecarga na rede induzida pela recolha de postais, assim como o custo associado ao seu processamento, foi extensivamente avaliada pelos autores do NetSight, podendo ser consultada em [4], pelo que omitimos aqui esta análise. Deixamos para trabalho futuro a avaliação da sobrecarga introduzida pela introdução de um intermediário entre o controlador e os comutadores, uma vez que essa avaliação só tem significado em redes reais, e não no ambiente simulado que usámos nesta avaliação.

Para avaliar a eficácia do NetSheriff na detecção de erros em SDNs, injectámos diferentes faltas nos equipamentos de modo a gerar os diversos tipos de erros enumerados na Tabela 1. Mais concretamente, para injectar as faltas nos comutadores alterámos as tabelas de encaminhamento através do comando

`sudo ovs-ofctl mod-flows <switch-id> <flow>`.

Desta forma, o NetSheriff não altera os grafos de propagação, uma vez que as regras não foram modificadas pelo controlador.

As experiências foram realizadas usando as topologias apresentadas na Figura 3. Usando estas topologias, e recorrendo a diferentes faltas, recriámos as diferentes categorias de erros listadas na Tabela 1.

Com excepção do caso da Figura 3d, todos os erros foram gerados tendo a mesma configuração de base. As ligações a azul formam os caminhos esperados para uma determinada classe de equivalência, instalados pelo controlador. As ligações a vermelho e tracejadas indicam caminhos que deveriam ser observados, mas que o comutador deixou de encaminhar pacotes nesse sentido. Por usa vez, as ligações a verde indicam que o comutador encaminha pacotes por portos que não era esperado.

A Figura 3 representa o caso em que não existe erros. As Figuras 3b-3f representam os casos particulares dos erros identificados na Tabela 1, pela mesma ordem: *b)* O comutador $D$ encaminha pacotes para $E$ quando não devia. *c)* O comutador $B$ deveria encaminhar pacotes apenas para $\{C, F\}$ mas encaminha para $\{C, F, G\}$. *d)* Existem divergências nos caminhos esperado e observado mas o pacote chega ao destino (sem duplicação). A ligação de $H$ para $G$ está ilustrada a azul porque essa ligação existe, apesar de não ser atingível por pacotes dessa classe de equivalência, pois não existem ligações ativas para $H$. *e)* O comutador $B$ deveria encaminhar pacotes para $\{C, F\}$ mas só encaminha para $\{F\}$. *f)* O comutador deixa $B$ de enviar para a totalidade dos portos esperados.

Os resultados das experiências mostram que NetSheriff foi capaz de detectar correctamente todos os cenários de erro testados. Realizámos também experiências adicionais, usando controladores POX e NOX inalterados em topologias lineares e *fat-trees* de vários tamanhos, para avaliar o funcionamento do NetSheriff

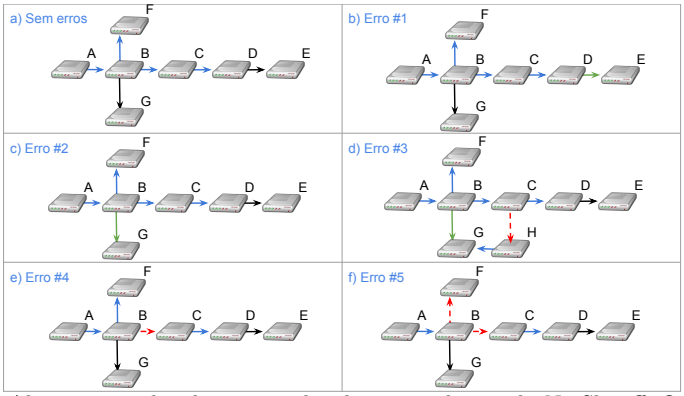

Figura 3: Alguns exemplos de erros utilizados na avaliação do NetSheriff. Os identificadores dos erros correspondem aos descritos na Tabela 1.

em diversas situações possíveis (por exemplo, alguns controladores não permitem comutadores ligados em ciclo, independentemente de se usar o NetSheriff ou não). O NetSheriff também funcionou correctamente nestas experiências.

O protótipo actual possui, no entanto, uma limitação que requer trabalho futuro. O NetSheriff gera um alerta sempre que existe uma disparidade entre os caminhos esperados e observados. No entanto, estas disparidades podem ocorrer devido a avarias dos equipamentos ou devido à perda de pacotes. Nas nossas experiências, o NetSheriff gerou ocasionalmente falsos positivos quando a rede foi sobrecarregada ao ponto de esgotar a largura de banda e criar a perda de pacotes. Distinguir correctamente estes dois cenários obriga a uma análise do caminho de vários pacotes, para perceber se o erro é determinista ou se só acontece esporadicamente. Por outro lado, isto abre a hipótese de usar o NetSheriff para detectar também outras classes de problemas, como ligações sobrecarregadas.

Note-se que os próprios postais também se podem perder. Este problema pode ser evitado distribuindo o coleccionador e assegurando que as rotas para os coleccionadores não se encontram sobrecarregadas. É também possível configurar o NetSheriff para só gerar uma alerta se a disparidade ocorrer para vários pacotes consecutivos.

## 6 Conclusões

Neste artigo propomos e avaliamos o NetSheriff, uma ferramenta de depuração automática de redes definidas por software. O NetSheriff combina técnicas de validação formal (para obter os caminhos esperados dos pacotes) e mecanismos de registo de pacotes (para obter os caminhos observados) com o objectivo de realizar uma análise diferencial, que permite identificar com exactidão qual o equipamento onde ocorreu a falha. Avaliámos experimentalmente o NetSheriff com diferentes tipos de erros, que exercitam diferentes aspectos da análise diferencial. Os resultados mostraram que o NetSheriff foi capaz de detectar os erros

nos equipamentos em todos cenários testados. Como trabalho futuro, pretendemos estender o NetSheriff para lidar com outros tipos de problemas, como a perda excessiva de pacotes devido, por exemplo, à congestão de uma ligação.

**Agradecimentos** Este trabalho foi parcialmente suportado pela Fundação para a Ciência e Tecnologia (FCT) e pelo PIDDAC através do projecto com a referência UID/-CEC/50021/2013. O software VeriFlow foi desenvolvido no Department of Computer Science at the University of Illinois at Urbana-Champaign.

## Referências

1. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: Vericon: Towards verifying controller programs in software-defined networks. SIGPLAN Not. 49(6), 282–293 (Jun 2014)
2. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A nice way to test openflow applications. In: NSDI. pp. 10–10 (2012)
3. Cisco Systems Inc.: Spanning tree protocol problems and related design considerations. `http://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/10556-16.html` (Aug 2005)
4. Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., McKeown, N.: I know what your packet did last hop: Using packet histories to troubleshoot networks. In: NSDI
5. Handigol, N., Heller, B., Jeyakumar, V., Maziéres, D., McKeown, N.: Where is the debugger for my software-defined network? In: HotSDN. pp. 55–60. ACM (2012)
6. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real time network policy checking using header space analysis. In: NSDI. pp. 99–112. USENIX Association, Berkeley, CA, USA (2013), `http://dl.acm.org/citation.cfm?id=2482626.2482638`
7. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: Veriflow: Verifying network-wide invariants in real time. In: NSDI 13. pp. 15–27. USENIX (2013)
8. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: Rapid prototyping for software-defined networks. In: Hotnets. pp. 19:1–19:6. ACM (2010)
9. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P.B., King, S.T.: Debugging the data plane with anteater. In: SIGCOMM
10. McKeown, N.: How SDN will Shape Networking. Available at `https://www.youtube.com/watch?v=c9-K5O_qYgA` (Oct 2011)
11. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38(2), 69–74 (Mar 2008)
12. Scott, C., Wundsam, A., Raghavan, B., Panda, A., Or, A., Lai, J., Huang, E., Liu, Z., El-Hassany, A., Whitlock, S., Acharya, H., Zarifis, K., Shenker, S.: Troubleshooting blackbox sdn control software with minimal causal sequences. SIGCOMM Comput. Commun. Rev. 44(4), 395–406 (Aug 2014)
13. Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A.: Ofrewind: Enabling record and replay troubleshooting for networks. In: USENIX ATC. pp. 29–29. USENIX Association (2011)