UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Pedro Langbecker Lima

ESTUDO DE ESTRATÉGIAS DE DETECTAÇÃO DE TRAPAÇAS EM JOGOS *ONLINE*

Santa Maria, RS
2017

Pedro Langbecker Lima

ESTUDO DE ESTRATÉGIAS DE DETECTAÇÃO DE TRAPAÇAS EM JOGOS *ONLINE*

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computaçãoda Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

ORIENTADOR: Prof. João Vicente Ferreira Lima

Santa Maria, RS
2017

Pedro Langbecker Lima

ESTUDO DE ESTRATÉGIAS DE DETECTAÇÃO DE TRAPAÇAS EM JOGOS *ONLINE*

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computaçãoda Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

Aprovado em 7 de novembro de 2017:

João Vicente Ferreira Lima, Dr. (UFSM)
(Orientador)

Carlos Raniery Paula dos Santos, Dr. (UFSM)
(Presidente)

Sérgio Luís Sardi Mergen, Dr. (UFSM)

Santa Maria, RS
2017

DEDICATÓRIA

Dedico este trabalho de conclusão de curso a meus companheiros Rafael Trindade, Vinícius Dreifke e Márian Pires, pelo apoio e incentivo.

AGRADECIMENTOS

Em especial, ao meu orientador prof. João Vicente Ferreira Lima pelos bons ensinamentos recebidos, por me apoiar e pelas oportunidades oferecidas que me possibilitaram aprender muito durante todo esse período.

Aos professores que compuseram a banca Carlos Raniery Paula dos Santos e Sérgio Luís Sardi Mergen pelas sugestões que contribuíram enormemente para melhorar este trabalho.

Ao curso de Ciência da Computação pelos infinitos conhecimentos adquiridos durante os quatros anos de estudo e à Universidade Federal de Santa Maria pela oportunidade de fazer parte de uma grande instituição pública que foi "meu lar" durante todo esse período.

À minha mãe Ulana Regina da Rosa Langbecker por ter me dado todas as condições para que eu chegasse até aqui, pelo carinho e amor que serviram de suporte para que eu conseguisse vencer as dificuldades que surgiram no decorrer do caminho. E ao seu companheiro Carlos Rosano por ser esta pessoa tão parceira e incentivadora de todas as horas.

Ao meu pai Paulo Ricardo Ferreira Lima (in memoriam) que me ensinou muito sobre o valor do trabalho, da perseverança e que sempre vibrou por minhas conquistas.

À Márian Pires por estar sempre ao meu lado, por ser uma companheira espirituosa e alegre.

A minha tia dinda Andrea Langbecker pelo apoio e revisões que me fizeram acreditar em dias melhores.

A todos os meus amigos e colegas que acreditaram em mim e me apoiaram para que eu concluísse mais essa etapa da minha vida!

A minha banda Shared Life, enfim, à música, que garantiram a minha (in)sanidade durante todo esse período de muito estudo!

placeholder

()

RESUMO

ESTUDO DE ESTRATÉGIAS DE DETECTAÇÃO DE TRAPAÇAS EM JOGOS *ONLINE*

AUTOR: Pedro Langbecker Lima
ORIENTADOR: João Vicente Ferreira Lima

Assim como em *softwares* no geral, a segurança é relevante para a qualidade dos jogos *online*. Um dos problemas a serem minimizados é a utilização de trapaças nos jogos, ato que pode ocasionar em prejuízos econômicos para empresas, assim como desestimular os consumidores do produto *online*. Um destes tipos de trapaças envolve a alteração do código fonte do cliente do jogo e/ou utilização de outra aplicação sobre o jogo que permite ao usuário modificar dados do cliente, ou até mesmo adulterações para acessar estados sensíveis do jogo que de outra forma estariam indisponíveis para o jogador. Estes problemas podem ser remediados com a utilização de diferentes estratégias e ferramentas, que podem ser otimizadas para cenários específicos ou se tornarem inviáveis pelo excesso de processamento. Este trabalho estuda diferentes métodos de validação de mensagens existentes, executando alguns dos métodos em um protótipo de jogo e avaliando suas características. Os resultados usando o SSIP mostraram-se eficientes no tempo de execução, enquanto o verificador com execução simbólica mostrou-se ineficaz, além de demonstrar dificuldade para verificar grandes quantidades de mensagens.

Palavras-chave: Cliente-servidor. Sistemas Distribuídos. Execução Simbólica. Jogos Online. Validação de Mensagens. Servidor de Auditoria

ABSTRACT

COMPARISON BETWEEN BEHAVIOUR VALIDATION METHODS IN ONLINE GAMES

AUTHOR: Pedro Langbecker Lima
ADVISOR: João Vicente Ferreira Lima

As in software in general, security is also relevant to online games quality. One of the faced problems is the use of cheats, which can cause economic loss for companies, as well discourage online product consumers. One of these cheats consists on modifying the game client source code and/or running a third party application over the game that allows the user to change client information, or even adulterations to access sensitive game states that usually are not available to the player. These problems can be corrected with the use of different strategies and tools, which can be optimized to specific scenarios or become unworkable due to over-processing. This work presents a study of different methods of validating existing messages, running some of the methods on a game prototype and analyzing their characteristics. The results of using SSIP proved to be efficient at run time, while the verifier with symbolic execution was considered inefficient, in addition to showing difficulties to verify massive messages.

Keywords: Server-client. Distributed Systems. Symbolic Execution. Online Games. Messages Validation. Audit Server

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

RTS	*Real-Time Strategy* (Estratégia em Tempo Real)
NVE	*Networked Virtual Environments* (Ambientes Virtuais em Rede)
DNS	*Domain Name System* (Sistema de Nomes de Domínios)
IA	Inteligência Artificial
MMO	*Massive Multiplayer Online* (Jogos Online para Multijogadores)
$SSIP$	*Secure Semantic Integrity Protocol* (Protocolo de Integridade Semântica Segura)
MAC	*Message Authentication Code* (Código de Autenticação de Mensagem)

SUMÁRIO

1 INTRODUÇÃO

Jogos *online multiplayer* são populares e lucrativos, mantendo-se sempre em crescimento na indústria de jogos. Em 2016, por exemplo, a taxa de crescimento econômico dos jogos *online* aumentou em 4.2%, com uma taxa de rendimento de 27% (Gráfico 1.1) do total de rendimento de todos os tipos de jogos. Parte deste crescimento deriva dos jogos MMO's (*Massive Multiplayer Online*), estilo de jogo onde grandes quantidades de jogadores jogam simultaneamente através da internet.

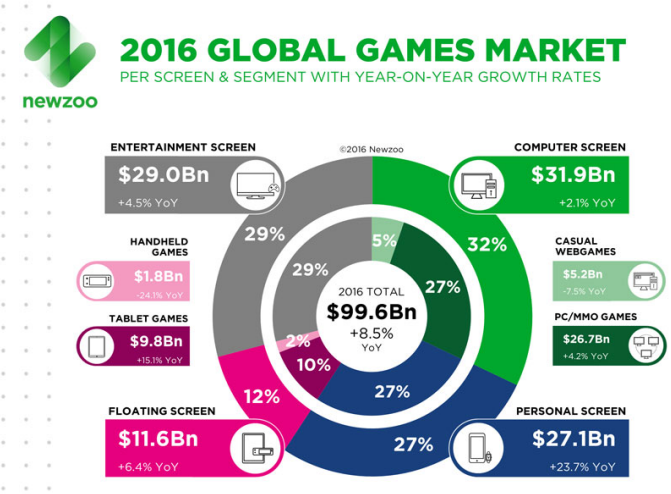

Gráfico 1.1 – Gráfico ilustrando rendimentos do mercado de jogos no ano de 2016.

Fonte: Adaptado de Newzoo (2016).

Porém, enquanto crescem e se tornam uma das aplicações mais populares para a Internet, as trapaças se tornaram um fenômeno chamativo no estado atual dos jogos na internet, atuando não só negativamente nas experiências dos outros jogadores como também como um problema de segurança para o jogo e a empresa responsável por ele (MCCREARY; CLAFFY, 2000).

Desde sua popularização, a indústria de jogos *online* vem sendo alvo de diferentes tipos de trapaças, que foram evoluindo e modificando-se ao longo dos anos. Muitas delas acabaram causando problemas financeiros para a empresa responsável do jogo. *Age of Empires* e *America's Army* são alguns exemplos de jogos *online* que sofreram perdas substanciais devido ao uso de trapaças por seus jogadores (SPOHN, Agosto 2016).

Jogos de *Networked Virtual Environments* (NVEs), reconhecidos pela grande imersão que trazem aos jogadores, seja pela qualidade gráfica ou ambientação de qualidade, acabam sofrendo muito com as trapaças. Como a sensibilidade para perceber atividades ilícitas se torna maior pela ambientação grande deste tipo de jogo, o impacto negativo aos jogadores acaba sendo ainda pior. As trapaças consequentemente tornaram-se uma das

formas mais rápidas de destruir a comunidade de um jogo e sua reputação comercial.

Sabendo do risco e dos problemas de segurança, diversos estudos e trabalhos surgiram para mitigar ou detectar diferentes formas de trapaças que se proliferam com a popularização dos jogos *online* no mercado. Entretanto, assim como alguns tipos de trapaças puderam ser inviabilizadas com boas escolhas de *design* e alterações mínimas, outras formas acabaram necessitando de recursos computacionais altos para serem mitigados. Relações de perda e ganho podem ser questionadas pelos criadores do *software*, onde pode-se trocar uma maior segurança do jogo por um desempenho e responsividade inferior.

Uma exemplo de vulnerabilidade que pode ser explorada pelos trapaceiros, por exemplo, é a adulteração do *software* que o cliente utiliza no serviço de jogo. Em um modelo cliente-servidor, em que o cliente interage com o servidor, solicitando serviços, e o servidor atende e responde as solicitações, se a mensagem for modificada de forma a tornar-se inválida de acordo com os padrões e regras impostas pelo servidor do jogo, o servidor deve perceber sua ilegitimidade. Caso a ação não seja averiguada no servidor como inválida, a integridade da partida pode ser comprometida. Trapaças como movimentações ilegais, atributos de personagem acima de seu devido valor, ou mesmo alterações em metadados da partida podem ocorrer devido a este tipo de trapaça. Outras trapaças, como a modificação do *driver* gráfico utilizado para remover as texturas de paredes, são exemplos comuns que podem ser encontrados hoje em dia.

Este trabalho tem como objetivo estudar diferentes comportamentos que podem ser adotados para previnir ou detectar trapaças em jogos *online*. No capítulo 2, como o espectro de trapaças é extensivo, foi estudado e catalogado os tipos de trapaças em diferentes conjuntos. O trabalho foca em um destes conjuntos de trapaças, abordando estratégias para detectar ou dificultar os tipos de trapaça encontrados neste conjunto. No capítulo 3, os diferentes métodos e estratégias são estudados e analisados, enquanto há um foco maior em dois métodos conhecidos na literatura: uso da execução simbólica para encontrar trapaças, e sistema de auditoria para verificar ações inválidas a partir da definição de estados abstratos e concretos.

O capítulo 4 apresenta a criação do jogo *online Shooterman*, implementado em C++, onde descreve-se a aplicação tanto do cliente quanto do servidor. Juntamente com o jogo, a implementação das duas estratégias sobre *Shooterman* é demonstrada. No capítulo 5, os resultados obtidos dos dois métodos implementados sobre o jogo *Shooterman* são avaliados e comparados.

2 TRAPAÇAS EM JOGOS ONLINE

2.1 DEFINIÇÃO DE TRAPAÇA

Segundo (YAN; RANDELL, 2005), no cenário de jogos *online*, qualquer comportamento que um jogador use para ganhar vantagem sobre outros jogadores ou alcançar um objetivo é considerado trapaça se, de acordo com as regras do jogo ou critério da operadora do jogo (fornecedora de serviços do jogo, que não necessariamente desenvolveu o jogo), a vantagem ou objetivo não deveria ser alcançada.

A partir da definição formal de uma trapaça no contexto de jogos *online*, é possível categorizar os diferentes tipos de trapaças encontrados, reforçando suas características e consequências. Esses difernetes tipos de trapaças abordados por ele podem ser agrupados em conjuntos que compartilham da mesma fonte do problema. Neste capítulo são abordados os diferentes conjuntos de trapaças existentes, dissertando sobre alguns exemplos e problemas encontrados ao longo da evolução dos jogos *online*. Os conjuntos classificados são divididos em diferentes categorias: adulteração do jogo, abuso das opções do jogo, segurança de rede, engenharia social e exploração da inteligência de máquina.

2.1.1 Adulteração do Jogo

Neste conjunto, o usuário trapaceiro utiliza tanto do *software* como do *hardware* para obter vantagens sobre os outros usuários.

A adulteração do código do jogo e/ou das configurações dos dados da aplicação do cliente, por exemplo, é um tipo de trapaça muito comum em jogos RTS (*Real-Time Stategy*), estilo de jogo onde os jogadores posicionam suas unidades e estruturas para controlar e avançar as áreas do mapa ao mesmo tempo que destroem com a do adversário. Neste cenário, a *Fog of war* (Névoa da Guerra) restringe a visão do mapa do jogador de acordo com suas tropas e construções, impossibilitando-o de reconhecer o posicionamento inimigo sem ter visão do local que ele se encontra. Com a utilização de um programa de terceiro, é possível filtrar os dados do cliente que informam o posicionamento das tropas adversárias, facilitando o reconhecimento do posicionamento adversário.

Outro tipo de trapaça, mais comum em jogos FPS (*First Person Shooter*), é o *Wall Hack*. Nos FPS, o jogador controla um personagem que possui acesso a armas de fogo e deve se movimentar pelo cenário, atirando nos adversários e evitando ser atingido. Como o ato de surpreender o adversário e reconhecer seu posicionamento é importantíssimo para este tipo de jogo, qualquer informação sobre o adversário obtida ilegalmente trás uma van-

tagem imensa para o jogador trapaceiro. Modificar a infraestrutura do cliente com um driver gráfico, por exemplo, pode ser usado para tornar as paredes de um jogo transparentes. No cenário de FPS, isto facilitaria a localização dos outros jogadores, que não seriam capazes de reconhecer o posicionamento do trapaceiro da mesma forma que ele os reconhece.

Ambos exemplos de trapaças, apesar de utilizarem ferramentas diferentes para conseguir vantagens sobre os adversários, são causados pelo mesmo problema encontrado no lado do cliente: a confiança excessiva sobre ele. Nos dois exemplos, informações relevantes sobre os adversários, como os posicionamentos dos jogadores ou tropas adversárias, situam-se não somente no servidor, mas também no cliente. Estes dados, se descobertos ou alterados, podem gerar benefícios.

2.1.2 Abuso das Opções do Jogo

As diversas opções que um jogo disponibiliza podem ser desbalanceadas se executadas em determinada ordem ou de determinada forma, sendo responsabilidade dos desenvolvedores do jogo buscar evitar estes tipos de abusos. Nos exemplos deste conjunto, as trapaças categorizadas comprometem o funcionamento ideal do sistema do jogo, pelo abuso das opções que o jogo dispõe.

O conluio, trapaça em que dois ou mais indivíduos combinam-se para ganharem vantagens desonestas, ocorreu no popular jogo StarCraft (BLIZZARD, 2017), onde dois jogadores arquitetavam uma aliança um com o outro. Cada um perdia uma partida para o outro no modo competitivo, e depois repetiam este processo indefinidamente. Como as derrotas que eles sofriam um do outro não reduziam os pontos de vitória do sistema, apenas as vitórias que obtinham um do outro eram computadas, elevando seus pontos e tornando o processo de subir de posição no jogo muito mais fácil, sem a necessidade de vitórias legítimas. Este tipo de trapaça é conhecido como *"win trading"*.

Outro tipo de trapaça semelhante, conhecido como Abuso dos Procedimentos do Jogo, é o ato de fugir da partida enquanto estiver em uma situação desfavorável. Em alguns jogos, pelo procedimento adotado no sistema, o ato de se desconectar da partida invalida a disputa, independentemente da situação em que ela se encontrava. Caso o sistema não esteja preparado para lidar como este tipo de situação, o jogador que se desconectou não sofre as consequências da derrota da partida, ou seja, se beneficia das falhas derivadas do procedimento adotado.

Alguns serviços de jogos, ou mesmo plataformas, como a plataforma Steam (STEAM, 2003), possibilitam que personagens ou itens virtuais adquiridos nos jogos sejam vendidos por dinheiro real para outros jogadores. Em uma trapaça com espólios virtuais, um *cheater* pode oferecer um item virtual, receber dinheiro real sobre ele de outro usuário, mas nunca entregar o item como combinado. Além de beneficiar o usuário que utiliza esta trapaça, o

prejuízo causado a vítima é financeiro.

Em alguns jogos RTS, um jogador trapaceiro pode atrasar seu próprio movimento até saber todos movimentos de seus adversários, ganhando assim uma grande vantagem. Esta forma de *look-ahead* (olhar para a frente) é um tipo de trapaça de tempo. (BAUGHMAN, 2001) Outro trapaça deste tipo é a *suppress-correct*, na qual o jogador obtém vantagens ao enviar propositalmente suas mensagens de *update* nos tempos "certos". (BAUGHMAN, 2001)

O uso abusivo de *bugs* ou *loopholes* (falhas no sistema que usuários podem explorar para ganhar uma vantagem injusta ou não intencional), por serem disponibilizados indiretamente, podem ser categorizados neste conjunto. Os usuários podem usufruir das falhas existentes conhecidas do sistema, sem precisar modificar o código do jogo ou obter dados com outros programas, para se favorecer em relação aos outros jogadores.

2.1.3 Segurança de Rede

Tratando-se dos problemas de segurança de redes tradicionais, pode-se tratar um paralelo com os jogos *online*. O envenenamento de cache DNS, por exemplo, pode ser comparado com um servidor de jogo falso. Se não existe um mecanismo adequado para autenticar o servidor do jogo na aplicação do cliente, um trapaceiro pode coletar várias dados das contas de usuários usuando um servidor falso.

Além do envenenamento de cache DNS, a análise de pacotes pode ser classificada como uma forma de trapaça neste cenário. Quando os pacotes da comunicação são trocados em formato de texto, algum usuário trapaceiro pode analisar os pacotes por meio de um analisador de *sniffer* e inserir, deletar ou modificar eventos do jogo ou comandos transmitidos via rede.

Um trapaceador também pode ganhar vantagens por negar serviços para seus jogadores *peer*. Por exemplo, um jogador pode atrasar as respostas de seu oponente sobrecarregando sua conexão. Outros jogadores *peer* poderiam supor que algo estaria errado com a conexão da vítima, concordando em removê-la da partida.

2.1.4 Engenharia Social

Usuários mal intencionados podem enganar outros usuários, fingindo serem administradores do jogo e requisitando informações pessoais de suas contas. Senhas geralmente são as chaves para grande parte ou todos os dados e autorizações que o usuário tenha no sistema do jogo. Caso sua senha seja comprometida, a vítima pode sofrer consequências como roubo de informações pessoais ou perda de conta.

2.1.5 Exploração da Inteligência de Máquina

Técnicas de inteligência artificial podem ser exploradas por um jogador em alguns jogos online. Por exemplo, o avanço da pesquisa do xadrez na computação produziu diversos programas que podem competir com humanos no nível mestre. Em uma partida online de xadrez, um usuário trapaceiro pode olhar os melhores candidatos para seu próximo movimento utilizando um programa de xadrez de computador. Isto se deve ao fato da superioridade, nesta situação específica, da inteligência artificial sobre um ser humano ordinário. Este tipo de trapaça pode existir em diversos outros jogos *online*, incluindo jogos de carta ou jogos de tabuleiro tradicionais, dependendo de suas propriedades do jogo - se o jogo pode ser modelado como um problema computável - e da maturidade de pesquisas de IA deste jogo. Um exemplo recente é a IA do jogo Go (GURU,), jogo de tabuleiro de origem chinesa, que teve sua IA em crescimento nos últimos anos. Em 2002, a IA mais forte de Go podia ser derrotada por um jogador humano amador (MüLLER, 2002), enquanto que em 2016, a IA criada pela empresa Google (GOOGLE. . . , 2017) venceu um dos melhores jogadores do mundo no jogo. (PRADO, 2016)

Nesta taxonomia apresentada a cerca das diversas trapaças existentes, o tipo abordado e analisado neste trabalho é focado no conjunto de trapaças representado como adulteração do jogo. As metodologias explanadas no Capítulo 3 trazem algumas soluções para a resolução destes problemas onde o cliente deliberadamente modifica sua versão do jogo ou suas mensagens na comunicação, podendo realizar ações fora das regras estabelecidas previamente.

3 MÉTODOS CONTRA TRAPAÇAS EM JOGOS ONLINE

Diferentes métodos podem ser empregados para evitar ou detectar trapaças que usuários mal intencionados estejam utilizando. Focando em um conjunto mais restrito de trapaças, mais especificamente, as apresentados no Capítulo 2.1.1, diferentes estratégias podem ser abordadas para se previnir. Dois destes métodos, abordados em estudos anteriores por (JHA et al., 2007) e (BETHEA; COCHRAN; REITER, 2008), são explicados nos Capítulos 3.4 e 3.5, e implementados nos Capítulos 4.1 e 4.2.

3.1 VERIFICAÇÃO DO EXECUTÁVEL

É notório que uma das formas de executar trapaças é alterar o executável do jogo, modificando dados de atributos do personagem, ou permitindo que determinadas ações inicialmente proibidas sejam possíveis de serem executadas. Uma das formas de evitar isso é checar periodicamente a conformidade do executável do usuário. Por exemplo, o cliente do jogo, antes de possibilitar que o usuário se conecte ao jogo com sua conta, utiliza um verificador no executável do usuário e o compara com um executável válido. Uma estratégia é a utilização de uma *hash* criptográfica, que permite a compressão unidirecional (compressão que não pode ser revertida) do executável do cliente, que é então enviado ao servidor e comparada com a *hash* previamente criada de um executável confiável. Se forem compatíveis, conclui-se que o usuário não alterou seu cliente de jogo. Esta estratégia, entretanto, ainda não resolve outros problemas encontrados em um cliente sem autoridade. Outros programas de terceiros ainda podem interagir com as informações que se encontram na aplicação, possibilitando que ações inválidas ocorram se o servidor não averiguá-las.

3.2 CLIENTE SEM AUTORIDADE SOBRE O ESTADO DO SERVIDOR

A estratégia adotada mais comum para se previnir de maus comportamentos dos clientes é garantir que o cliente não possua estado de autoridade que possa afetar o servidor ou a aplicação. O básico dessa abordagem é enviar ao servidor todos *inputs* do cliente, que são executados e validados diretamente no servidor. Esta estratégia não se preocupa com o processamento extensivo realizado no servidor, afinal, nenhum processamento crítico é executado no computador do cliente. Pode-se categorizar este tipo de estratégia como inviável para aplicações com custos computacionais significativos ou com público

alvo grande, o que é o caso para a maioria dos jogos *online* encontrados atualmente. Para jogos onde o tempo de resposta não é impactante na experiência proporcionada aos jogadores, como em jogos de turnos de estratégia, esta opção pode ser considerada viável. Jogos de xadrez *online* ou jogos de cartas, por exemplo, são estilos de jogos que não são afetados tão negativamente pelo tempo de resposta maior do servidor.

3.3 VALIDAÇÃO POR OUTRO CLIENTE

O servidor pode agir como o meio de comunicação entre dois clientes, fazendo com que outro jogador valide a mensagem enviada. Nesta estratégia, a validação ocorre diretamente na aplicação de outro usuário que retorna ao servidor se a ação é considerada válida ou não. Apesar do custo computacional acontecer diretamente no cliente, e na maioria dos casos, esse processamento ser insignificante isoladamente e rápido, o número de mensagens de uma ação é dobrada. Para cada mensagem enviada do cliente ao servidor, outras duas novas mensagens devem ser transmitidas, do servidor ao cliente dois, e novamente do cliente dois ao cliente um. Outro problema é a divulgação de informações que o jogador poderia não ter acesso. Em uma situação onde o cliente 2 recebe informações do cliente 1 para validar, ele acaba ganhando acesso aos dados a serem verificados. Em cenários onde existem informações confidenciais a outros jogadores, esta estratégia não é muito útil, pela necessidade da quebra de confidenciabilidade sobre os dados a serem verificados.

3.4 VALIDAÇÃO COM EXECUÇÃO SIMBÓLICA

A execução simbólica é uma das formas de analisar um programa e determinar quais entradas dele causaram a execução de cada parte do programa. Em vez de utilizar valores de entrada concretos, como números inteiros ou de ponte flutuante, os valores são transformados em valores simbólicos, e, como eles, sua saída é gerada simbolicamente. Seu uso mais convencional é no teste de *software*, utilizado na análise de erros. Nessa análise, as entradas e condições que acarretaram a ocorrência dos erros são previstas utilizando a execução simbólica.

A execução simbólica utiliza o paradigma Programação com Restrições, que consiste em especificar quais critérios (definidos formalmente como restrições) uma solução deve cumprir. Um resolvedor de restrições então utilizará as restrições especificadas juntamente com o fluxo lógico do programa para descobrir quais valores concretos de entrada gatilharam a ocorrência dos erros.

A Ilustração 3.1 representa o fluxograma gerado após a execução simbólica de *x*. No exemplo, verifica-se se *x* é maior que zero, e depois, se *x* é maior que dez. No primeiro teste, dois fluxos de execução podem acontecer: ou *x* é maior que zero, ou não. Em cada um destes novos fluxos de execução, é verificado se *x* é maior que dez. No segundo fluxo, a hipótese de *x* ser maior que dez é descartada, afinal, *x* mostrou-se menor que zero no teste anterior.

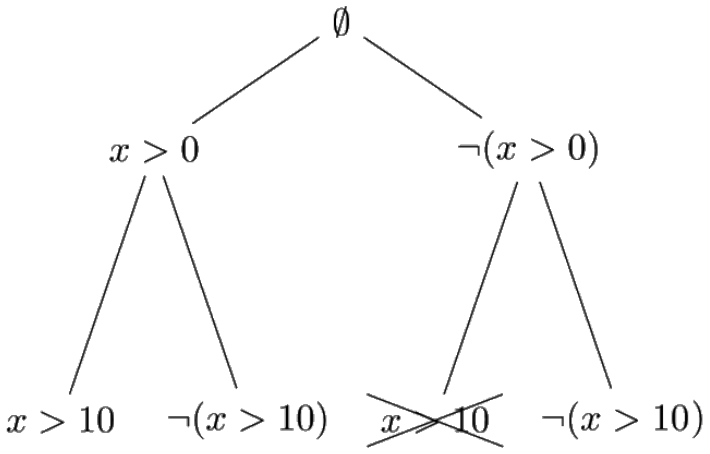

Ilustração 3.1 – Exemplo de execução simbólica de um programa simples.

Fonte: Retirado da apresentação de Collingbourne, Cadar e Kelly (2014).

(BETHEA; COCHRAN; REITER, 2008) demonstra um exemplo prático de uma implementação simbólica em um jogo simples.

```
100: loc ← 0;
101:
102: enquanto true faça
103:     key ← lekey();
104:     se key = ESC entao
105:         fimjogo();
106:     senao se key = '↑' entao
107:         loc ← loc + 1;
108:     senao se key = '↓' entao
109:         loc ← loc – 1;
110:     fim se
111:     envialocalizacao(loc);
112: fim enquanto
```

(a) Exemplo simples de um cliente

```
200: loc_ant ← 0;
201: loc ← loc_ant;
202: enquanto true faça
203:     key ← simbolico;
204:     se key = ESC entao
205:         fimjogo();
206:     senao se key = '↑' entao
207:         loc ← loc + 1;
208:     senao se key = '↓' entao
209:         loc ← loc – 1;
210:     fim se
211:     breakpoint;
212: fim enquanto
```

(b) Exemplo instrumentado para executar simbolicamente

Figura 3.1 – Exemplo de execução simbólica com código.

Fonte: Adaptado de Bethea, Cochran e Reiter (2008).

Neste exemplo simples do jogo Pong (PONG, 2017) da Ilustração 3.1, a aplicação do cliente lê a todo instante as teclas pressionadas pelo usuário, atualizando a posição do jogador de acordo com a direção que a tecla pressionada. Em cada iteração do laço principal do programa, o servidor também é atualizado, recebendo a posição atual que o cliente se encontra.

Ja na versão simbólica do programa, a nova variável *loc_ant* é instanciada como uma variável simbólica, assim como a variável existente *key*. Um *breakpoint* é criado substituindo a função *envialocalização*, possibilitando que sempre que o fluxo do programa alcançá-lo, se obtenha as restrições geradas em todas variáveis simbólicas. Percebe-se que o envio da mensagem ao servidor não ocorre na versão simbólica, afinal, ela não visa ser uma versão executável tradicional da aplicação, mas sim uma versão que simula os fluxos de operações realizadas durante uma execução.

Para cada escolha gerada na execução do programa, um novo ramo é criado juntamente com uma nova restrição. Essas escolhas são criadas no laço principal, em cada uma de suas iterações. Consequentemente, é possível construir um conjunto de restrições, que representam quais possíveis restrições podem ser geradas em cada iteração do laço principal. A partir do exemplo da Figura 3.1, dada a iteração *j*, define-se que o conjunto de possíveis restrições que podem ocorrer nessa iteração C_j é o conjunto

(loc = loc_ant + 1) $\vee$ *(loc = loc_ant – 1)* $\vee$ *(loc = loc_ant)*

Como pode ser deduzido, cada elemento do conjunto C_j depende dos valores anteriores atribuídos a variável *loc_ant*. Considerando um cenário onde o servidor sabe a última posição do cliente, por exemplo, três possíveis restrições podem ser verificadas na próxima mensagem que o cliente enviar. Caso nenhuma dessas restrições for atendida, ou seja, a variável recebida não for igual, um a mais ou um a menos que o valor anterior, é dedutível que a mensagem enviada pelo cliente é falsa e ele está trapaceando.

3.5 PROTOCOLO DE INTEGRIDADE DE SEMÂNTICA SEGURA

Esta abordagem utiliza um procedimento de auditoria, que é executado por um Servidor de Auditoria. Sua estratégia utiliza o conceito de estados abstratos e concretos na troca de mensagens entre as ações de um cliente. A partir dos estados dos clientes, é utilizado um algoritmo para verificar a integridade da sequência destes estados com a utilização de processos de auditoria.

3.5.1 Definição de Estados

Este método utiliza o conceito de estados abstratos e concretos, que são ilustrados na Figura 3.2. No exemplo que a Figura 3.2 traz, duas aproximações foram pensadas para checar se em um determinado vetor *a*, a posição *i* do vetor, *a[i]*, excede os limites do vetor. No primeiro modelo criado, existe um conjunto com todos possíveis valores que *i* pode conter, enquanto que, no segundo modelo, existem possíveis intervalos de valores possíveis que podem conter *i*. Nesta representação simples, é preferível utilizar o segundo modelo, pela sua maior abstração dos valores que *i* pode obter. A primeira aproximação utiliza valores concretos para a resolução do problema, enquanto a segunda utiliza valores abstratos. As setas verdes da figura representam as funções de transformação de domínio que ocorrem de um modelo para o outro. A transformação δ representa uma função de abstração (transposição do domínio mais concreto para o mais abstrato), e γ representa o processo inverso, uma função de concretização (transposição do domínio mais abstrato para o concreto).

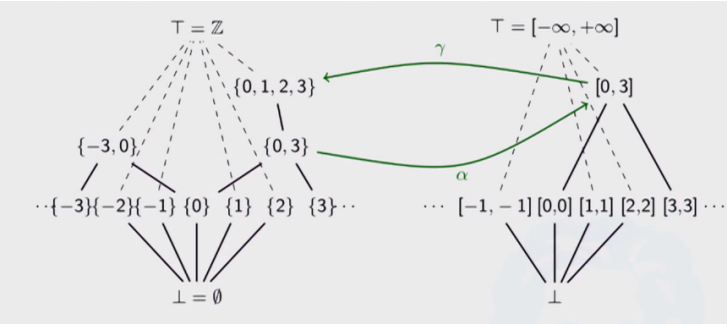

Ilustração 3.2 – Exemplo da função de Abstração e Concretização.

Fonte: Retirado da apresentação de Reichenbach (2015).

Formalmente, define-se que a partir de um estado abstrato *s*, γ (s) representa o conjunto de possíveis concretizações que podem ocorrer a *s*. Já um estado concreto S, sua função δ(S) corresponde a um único estado abstrato. Em outras palavras, um estado abstrato pode representar diferentes estados concretos, enquanto que um estado concreto representa um único estado abstrato. Alterações de um estado ocorrem constantemente, e são representadas por Δ, ou *diff*, sendo ele a mudança sofrida de um estado para outro. Em outras palavras, o estado S atualizado para o estado S' sofre a diferença Δ, portanto, S' $=$ S + Δ.

As funções de transformação de domínio γ e δ podem ser aplicadas não somente nos estados, mas também nas diferenças entre estados. $\gamma(\delta)$, por exemplo, expressa a abstração de uma *diff* δ qualquer, enquanto δ representa a concretização da *diff* δ. Algumas conclusões podem ser estabelecidas sobre as *diff*, como γ(S') $= \gamma$(S) + $\gamma(\delta)$ se S'$=$ S + Δ. Sendo α a representação de uma abstração de uma *diff* Δ aplicada ao estado S,

pode-se deduzir que $\Delta \in \gamma$(S, $\alpha(\Delta)$) e para cada $\Delta' \in \gamma$(S, $\alpha(\Delta)$) obtém-se α(S + Δ') = α(S´).

3.5.2 Aplicação dos estados

No Protocolo de Integridade de Semântica Segura (SSIP), representam-se as informações do servidor e do cliente em estados, que podem ser tanto abstratos quanto concretos. A representatividade destes estados pode ser encontrada na implementação realizada, referenciada no Capítulo 4.

Neste protocolo, o servidor principal da aplicação é denominado Servidor de Estados, que equivale a um servidor convencional de uma NVE, implementado para interagir com os usuários e estabelecer comunicações e trocas de dados sobre as informações do jogo. O servidor de estados contém um estado central abstrato ASTATE, que contém informações relevantes de todos clientes. Imaginando o estado ASTATE como um vetor, por exemplo, cada uma de suas partes é relevante para um cliente conectado a ele. Em outras palavras, de todo estado ASTATE, apenas a parcela ASTATE[cli_i] é relevante para o cliente cli_i.

Como os clientes possuem estados concretos das informações do jogo, e o servidor possui apenas uma versão abstrata disso, as funções de transposição de domínios apresentadas no capítulo 3.5.1 acabam ocorrendo constantemente na comunicação cliente-servidor. Exceto na primeira mensagem de inicialização do cliente com o servidor, por exemplo, todas mensagens posteriores enviadas pelo cliente passam pela função de transposição de abstração. Estas abstrações das mensagens do cliente existem para amenizar o excesso de processamento gerado no servidor nos testes que executa para verificar a validade da mensagem.

Um exemplo simplificado deste caso encontra-se no jogo implementado *Shooterman* (capítulo 4.0.1). Enquanto variáveis como posição são mantidas no servidor, variáveis de controle mais refinado das ações do personagem são mantidas unicamente no cliente de jogo, como por exemplo, o tempo de recarga de suas ações. Essas informações do estado do jogador são mais abstraídas para o servidor, enquanto são concretas para o cliente do jogador.

Em cada um dos ciclos do cliente, ele envia ao servidor uma abstração $\gamma = \alpha(\Delta)$, que contém uma abstração da *diff* dos últimos estados. Esta abstração é denominada *diff de requisição*. Após o recebimento da *diff* de requisição, o servidor válida a mensagem de acordo com suas semânticas pré-estabelecidas e retorna ao cliente uma confirmação das mudanças requisitadas e das atualizações realizadas pelos outros clientes conectados a NVE.

Utilizando esta técnica, um cliente malicioso ainda pode violar as regras semânticas

estabelecidas pelo servidor, pela susceptibilidade do protocolo, afinal ele só pode verificar se as atualizações abstratas são consistentes com o estado abstrato, sem levar em consideração os valores concretos avaliados. Para solucionar esta falha, ciclos de auditoria são utilizados constantemente para verificar as atualizações de acordo com os estados concretos dos clientes conectados.

3.5.3 SSIP

Outro servidor, denominado Servidor de Auditoria, é utilizado para verificar a integridade semântica dos estados que são constantemente atualizados pelos clientes. Este servidor requisita a um cliente sua sequência de atualizações de estados concretos de um tempo específico junto com seu estado concreto inicial completo, e a partir deles, computa os estados em ordem, verificando sua integridade.

Uma série de operações devem ser seguidas na comunicação do cliente com o servidor de auditoria, e (JHA et al., 2007) dividiu estas operações em três diferentes protocolos, que são apresentados a seguir. Nas ilustrações dos protocolos, são utilizados as setas $\rightarrow$ que representam pacotes enviados de forma não-confiável, e as setas $\hookrightarrow$, que representam pacotes enviados de forma confiável.

3.5.4 Protocolo de Inicialização

Este protocolo ocorre quando o cliente inicia seu contato com o servidor. A Figura 3.2 mostra a sequência de passos que deve ser tomada no início da comunicação do cliente com o servidor do jogo.

1. cli inicializa $t := 0$ e envia requisição de solicitação ao servidor de estado
2. Servidor de estado $\rightarrow$ servidor de auditoria : $k :=$ GeraMac(1^n)
3. Servidor de auditoria $\rightarrow$ cli : $h :=$ GeraHash(1^n)
4. Servidor de estado escolhe o estado S do conjunto ASTATE
5. Servidor de estado $\rightarrow$ cli: AuthMsg(k, S $\parallel n_0$, cli)
6. cli define S_0 como S.
7. cli $\hookrightarrow$ servidor de auditoria: $Q_0 := CFHash_h(S_0)$

Fonte: Traduzido de (JHA et al., 2007). Protocolo Initialize

Figura 3.2 – Definição do Protocolo de Inicialização.

Primeiramente, o cliente inicializa a variável *t*, que representa o número do ciclo que ele se encontra. No passo 2, o servidor de estado envia ao servidor de auditoria a variável *k*, que representa uma chave criptográfica. Essa chave criptográfica é gerada usando o *Message Authentication Code* (MAC), algoritmo que recebe como entrada uma chave secreta e uma mensagem a ser autenticada, gerando uma MAC, ou etiqueta.(MAC,)

No terceiro passo, define-se *h*, índice que representará qual função *hash* o cliente utilizará posteriormente em suas mensagens enviadas a auditoria. O parâmetro *n* da função GeraHash representa o nível de segurança da *hash*.

Em seguida, o servidor envia uma mensagem ao cliente contendo o estado S escolhido no quarto passo, e o cliente a define como seu estado primário S_0. A mensagem é enviada junto com a etiqueta MAC gerada.

No começo de cada auditoria, o cliente envia ao servidor de auditoria uma *hash* de seu estado concreto completo. Apesar do custo computacional ser relativamente alto dependendo do tamanho da informação contida em um estado concreto, a mensagem precisa chegar entre os próximos *n* ciclos do cliente. O último passo deste protocolo demonstra o envio do estado concreto S_0 ao servidor de auditoria. Após o envio, a inicialização é finalizada.

3.5.5 Procotolo de Atualização dos Status

Este protocolo é executado em cada ciclo do cliente. Ele consiste na atualização do estado do cliente após a confirmação do servidor,

Além de manter seu estado concreto atual, o cliente mantém seus estados anteriores em um *buffer* local, contendo até três estados completos e 3*n diffs* geradas ao longo de sua execução. Enquanto novos estados e suas *diffs* são inseridas no *buffer*, seus conteúdos antigos podem ser removidos por não serem mais úteis no protocolo. Essas características do *buffer* correspondem a um processo semelhante ao de uma janela deslizante, onde as informações relevantes que o *buffer* armazena são sempre as mais recentes. Todas mensagens recebidas do servidor de auditoria também são armazenadas no cliente de acordo com o intervalo de tempo determinado pela janela deslizante de seu *buffer*.

No início de cada atualização, o cliente gera a mudança Δ de acordo com sua ação, gerando uma abstração δ que é então enviada ao servidor de estado. Ele computa esta atualização a partir da abstração recebida, atualizando seu ASTATE. Após isso, ele envia uma mensagem de confirmação ao cliente com sua etiqueta MAC para identificar sua autenticidade como servidor.

Confirmada a mudança Δ gerada pelo cliente, ele atualiza seu novo estado, agora sendo S_{t+1}, como o passo 5 representa. A partir do tipo de *hash* definido no protocolo

1. cli computa a mudança Δ_{t+1} e gera uma nova abstração δ_{t+1}
2. cli $\rightarrow$ servidor de estado: δ_{t+1}
3. Servidor de estado computa a atualização δ'_{t+1} e atualiza ASTATE
4. Servidor de estado $\rightarrow$ cli: M_{t+1} := AuthMsg(*k*, $\delta_{t+1} \parallel n_t$ + 1, cli)
5. cli armazena Δ_{t+1} e computa $S_{t+1} = S_t + \Delta'_{t+1}$
6. cli $\hookrightarrow$ servidor de auditoria : $D_{t+1} := Hash_h(\Delta'_{t+1})$
7. cli incrementa variável *t*
8. caso *t* mod *l* $= 0$
   (a) cli deleta todos *diffs* Δ'_{t-i}
   (b) cli armazena S_t e começa a computar $Q_t := Hash_h\ (S_t)$

Fonte: Traduzido de (JHA et al., 2007). Protocolo StatusUpdate

Figura 3.3 – Definição do Protocolo de Atualização dos Status.

anterior de atualização realizado anteriormente declarado na variável *h*, o cliente gera um *hash* dessa mudança Δ e envia ao servidor de auditoria. O cliente incrementa o número de ciclos armazenado em *t*, e verifica se *t* é múltiplo de *l*. Se for, ele remove todas *diffs* mais antigas geradas anteriormente, além de armazenar seu estado concreto, computar a *hash* deste estado, e enviá-la ao servidor de auditoria.

3.5.6 Procotolo de Auditoria

É no protocolo de auditoria que se verifica a validade de um cliente em um certo período de tempo onde ele envia suas mensagens. O primeiro passo, como observado na Figura 3.4, é enviar uma mensagem ao cliente convocando-o a uma auditoria, juntamente com o índice t_0 que representará o índice inicial das ações que será enviada.

Quando ciente da abertura da auditoria, o cliente computa t_a. Este valor representa o índice do ciclo mais recente que ele começará a enviar. Então o cliente envia as *diffs* das mensagens de t_a até t_0. Em outras palavras, todas mudanças que ocorreram em seu estado deste o instante t_0 (início da auditoria) até o instante t_a (representado pela fórmula do passo 2) são enviadas ao servidor de auditoria. Além das *diffs*, seu estato concreto também é enviado, juntamente com todas mensagens M_i no mesmo intervalo de *a* a *0*. Estas mensagens são as mensagens de confirmação recebidas pelo servidor de estado no protocolo de Atualização dos Status (vide Figura 3.3, passo 4).

Em um exemplo, supondo que *l* seja 10, ou seja, a cada 10 ciclos do cliente, uma

1. Servidor de auditoria $\rightarrow$ cli : *audit* $\| \ t_0$
2. cli computa $t_a = \lfloor \ \frac{t_0}{l} - 2 \ \rfloor \ l$
3. cli $\rightarrow$ servidor de auditoria : $S_{t_a} \ \| \ \Delta'_{t_a}$ + 1 $\|$ ... $\| \ \Delta'_{t_o} \ \| \ M_{t_a}$ + 1 $\|$ ... $\| \ M_{t_0}$
4. Para $i = t_a$, ... $t_0 - 1$ onde $\hat{S}_{t_a} = S_{t_a}$ o servidor de auditoria computa $\hat{S}_{i+1} = \hat{S}_i + \Delta'_{i+1}$
5. Para todo $i = t_a +$, ..., t_0 o servidor de auditoria determina se Δ'_i está de acordo com as regras estipuladas
6. Para todo $i = t_a +$, ..., t_0, o servidor de auditoria verifica o MAC e a Hash enviadas pelo cli.
7. Servidor de auditoria compara as *hashs* S_{t_a} com Q_{t_a} e $\hat{S}_{t_a}$ + l com Q_{t_a} + l.

Figura 3.4 – Definição do Protocolo de Auditoria.

Fonte: Traduzido de (JHA et al., 2007). Protocolo Audit

nova *hash* do estado concreto é gerada e enviada ao servidor de auditoria. Na abertura da auditoria, o servidor de auditoria envia ao cliente t_0 equivalente a 30. A partir disso, o cliente calcula t_a, como no passo 2, resolvendo que t_a = 10. O cliente então envia seu estado concreto do ciclo t_a, juntamente com todas *diffs* do ciclo 10 ao 30, e todas mensagens de confirmação M do ciclo 10 ao 30. O servidor de auditoria então descobre todos estados concretos do ciclo 10 ao 30, usando as abstrações recebidas (passo 4). O servidor também verifica se as *diffs* são consistentes com as abstrações δ'(passo 5).

Com isso, usando as mensagens recebidas anteriormente Q_i (vide Figura 3.2, passo 7), e D_i (Figura 3.3, passo 6), o servidor de auditoria consegue comparar as abstrações recebidas aplicando-as a Q_i e verificar se equivalem as ações concretas D_i. Depois, ele verifica tanto o MAC contido nas mensagens M_i com *i* variando de t_a a t_i, quanto as *hashs* das *diffs* Δ' com as textithashs recebidas anteriormente em D. Depois o servidor de auditoria compara as *hash* dos estados concretos inicial e final recebidas com os estados gerados atráves do passo 4 e transformados em *hash* para esta comparação.

4 DESENVOLVIMENTO

Antes de implementar os métodos analisados, é necessário definir qual cenário será considerado e qual aplicação será alvo dos métodos. Este capítulo tem como base definir este modelo de jogo e especificar quais atributos foram levados em consideração na produção do *software*. Além disso, este capítulo demonstra como o jogo foi modelado, e como os métodos foram construídos a partir do cenário estabelecido.

A definição formal de um jogo pode ser compreendida como qualquer atividade que exiga um jogador e regras que devem ser seguidas. A definição de um jogo *online*, por sua vez, pode ser interpretada como um jogo que pode ser jogado utilizando uma rede de computador, possibilitando com que dois ou mais jogadores participem simultaneamente de uma partida mesmo estando em lugares diferentes. Este conceito pode ser aplicado para qualquer *software* de jogo, tanto *single-players* (jogos eletrônicos para um jogador) como *online*.

A definição utilizada neste trabalho consiste em uma aplicação que utiliza troca de mensagens entre cliente e servidor, em que o cliente interage com a aplicação por meio de suas ações no jogo. As ações escolhidas são enviadas ao servidor e informadas aos outros jogadores que estão conectados na partida. Para simplificar a construção do *software* e pela irrelevância para os métodos estudados, o ambiente virtual e gráfico do jogo foi desconsiderado em sua implementação.

4.0.1 Jogo

O jogo projetado e implementado é um jogo de tiro (mais conhecido no cenário de jogos como *shooter*), subgênero de jogos de ação, chamado de *Shooterman*, em que o foco principal do jogo se encontra nas ações que o personagem executa usando algum tipo de arma. No jogo, cada personagem possui uma arma de longa distância, com a qual pode atirar outros jogadores, derrotando-os, e assim, adquirindo pontos de vitória.

Apesar de *Shooterman* não possuir interface visual, é importante salientar que ele é um jogo com perspectiva *top-down*, estilo de jogo em que a câmera encontra-se acima do mapa e o jogador visualiza o cenário de uma perspectiva superior. Esta distinção é importante para a implementação adequada da movimentação dos jogadores e dos projéteis que irão atirar. Afinal, suas coordenadas devem condizer com a perspectiva da câmera do jogo.

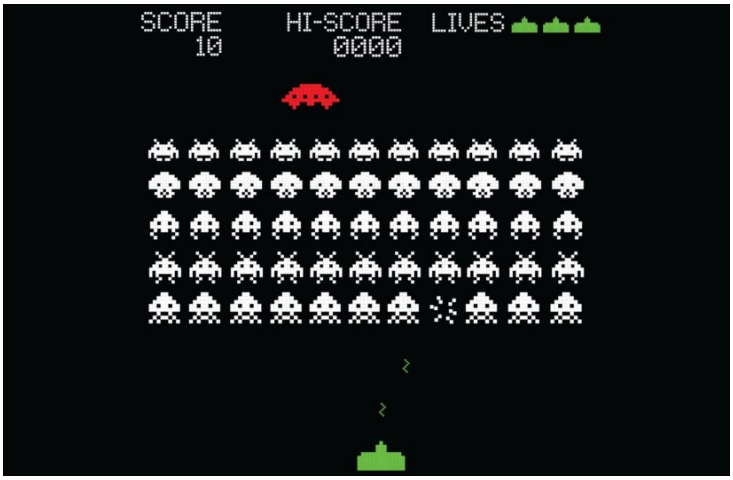

Figura 4.1 – Gameplay do clássico Space Invaders - Exemplo de jogo com câmera top-down.

4.0.2 Ações do Jogador

Cada jogador controla um único personagem, que pode atirar projéteis, bloqueá-los e movimentar-se pelo mapa.

O jogador pode movimentar-se em um cenário bidimensional em quatro direções ortogonais - esquerda, direita, baixo e cima. Sempre que ele se movimenta para uma direção, a direção atual que ele está olhando é atualizada. Esta direção será a direção que os projéteis que ele atirar irão seguir.

A ação principal do personagem é o seu tiro. Quando atira, um projétil é criado a partir de sua posição e atravessa o cenário até colidir com algum objeto ou exceder seu deslocamento máximo. Sempre que o personagem atira, ele não pode atirar novamente por um determinado intervalo de tempo - sua habilidade encontra-se em tempo de recarga.

O personagem pode também ativar um escudo, uma manobra defensiva, que dura um determinado período de tempo, previnindo o jogador de ser atingido por ataques inimigos. Este escudo também possui um tempo de recarga que deve ser respeitado.

Nenhuma dessas ações pode ser executada em um mesmo intervalo de tempo, ou seja, movimentar-se enquanto atira ou atirar enquanto utiliza seu escudo são ações inválidas de acordo com as regras estabelecidas.

4.0.3 Implementação

Duas aplicações foram implementadas na linguagem C++ versão 11: a versão do cliente e a versão do servidor. Na implementação de *Shooterman*, a autenticidade das mensagens, tanto do cliente quanto do servidor, não foram levadas em consideração da implementação dos métodos. Apesar da transmissão de pacotes utilizar o protocolo UDP,

a falha do envio de pacotes também foi descartada para simplificar a implementação dos métodos e avaliar suas funções primárias.

Todas ações executadas pelos jogadores foram enviadas do cliente ao servidor em um *json*. Foi utilizada a biblioteca *JSON for Modern C++*, para facilitar a criação de *jsons* e sua desconstrução em dados brutos utilizados na aplicação (LOHMANN, 2017).

```
{
  "player_id" : 4,
  "position": {
    "x" : 5,
    "y" : 8,
    "side" : 0
  },
  "shoot" : "false",
  "barrier_on" : "true"
}
```

Ilustração 4.1 – Estrutura json utilizada

A Figura 4.1 representa o conteúdo dos dados enviados ao servidor. O mesmo padrão é utilizado pelo servidor ao repassar a mensagem para os outros jogadores, que recebem o arquivo e atualizam as informações do jogo de acordo os dados recebidos. A variável *position* é responsável por representar a posição geográfica do personagem e o lado que ele se encontra. *Shoot* e *barrier_on* são booleanos de controle que verificam se o jogador está atirando ou ativando sua barreira. *Player_id* faz referência ao id do jogador que está realizando sua ação.

No servidor, a classe principal *Game* verifica a cada instante de tempo se um novo cliente conectou-se na aplicação. Sempre que uma nova conexão inicia-se, uma instância da classe *Connection* é criada, onde ela dispara uma *thread* para processar a nova comunicação cliente-servidor, enquanto paralelamente a *thread* principal continua averiguando por novas conexões. A Figura 4.2 representa a conexão utilizando a classe ServerSocket. Caso o server aceite o novo socket, uma conexão é criada e adicionada a lista de conexões, e uma *thread* é disparada na função newPlayerConnection().

Logo quando uma conexão é estabelecida, o servidor envia ao cliente a posição de seu personagem, junto com seu id que será utilizado para identificar seu personagem dentre os outros. Nota-se que o identificador deve ser único para que haja integridade no estado do servidor. Em uma versão mais refinada, em que os clientes cadastram-se na aplicação e possuem seus dados registrados em um banco de dados do servidor, a identificação torna-se mais legítima, afinal, os jogadores teriam seus identificadores únicos registrados junto com informações de suas contas, e utilizariam o mesmo identificador para diversas partidas realizadas na aplicação. Na versão atual, entretanto, os identificadores são provisórios e só se limitam a uma unica conexão.

```
while (true){
    ServerSocket* new_sock = new ServerSocket();
    server->accept (*new_sock);

    try {
        Connection* con = new Connection(this, new_sock, client_id);
        client_connections.push_back(con);
        cout << "Cliente " << client_id  << " conectado!" << endl;
        con->newPlayerConnection();
        client_id++;
    }
    catch (SocketException&) {
        cout << "Erro na conexao com cliente" << endl;
    }
}
```

Ilustração 4.2 – Thread principal do servidor

A classe *Connection* é responsável pela comunicação com o cliente conectado, com sua *thread* recebendo mensagens do cliente e retornando suas requisições. Sempre que uma nova mensagem é recebida, ela é validada e enviada a todos jogadores se for considerada semanticamente correta. Percebe-se que dependendo da implementação utilizada no servidor, a função *msgIsValid* varia. Em um cenário que o cliente não possui nenhuma autoridade sobre suas ações, a função avaliaria todas condições necessárias para a ação ser executada, checando sua integridade, e permitindo a ação caso a considerasse legítima. Na versão base da aplicação, entretanto, só é avaliado se o personagem que está realizando a ação realmente é o mesmo controlado pelo usuário da conexão, a partir de seu identificador id. A Figura 4.3 ilustra a função de comunicação, com a *thread* executando as operações da função update.

No cliente, a classe principal da aplicação que controla basicamente seu fluxo de operações também é a *Game*. Todos *inputs* do usuário, bem como as comunicações existentes com o servidor operam nesta classe, semelhantemente a classe *Connection* do servidor. Em cada ciclo de execução do cliente, a classe *Game* cuida para que todas suas informações sejam enviadas ao servidor, e que a resposta dele juntamente que contém informações de outros clientes sejam interpretadas e executadas na aplicação.

Sempre que o jogador pressiona uma tecla para atirar, e o cliente pode atirar, a classe *Game* garante que essa informação seja submetida ao servidor. Essas ações e as verificações que a própria aplicação do cliente faz são executadas na classe *Player*. Esta classe basicamente avalia as ações executadas pelo cliente, e possibilita suas execuções. A classe, por exemplo, verifica se o jogador pode usar seu escudo, ou se ele se encontra indisponível por ter sido utilizado recentemente. Algumas funções e variáveis importantes são ilustradas na Figura 4.4.

Todos objetos existentes no jogo, ou mesmo novos objetos que podem ser adiciona-

```
void Connection::update(){
  while (true){
    std::string received\_msg;
    *(server_socket) >> received_msg;

    *(player_log) << received_msg << endl;

    auto msg\_json = json::parse(received_msg);
    if (msgIsValid(msg_json) == true){
      updatePlayer(msg_json);
      game->sendToClients(msg_json);
    }
    else{
      cout << "usuario invalido" << endl;
    }
  }
}
```

Ilustração 4.3 – Função update da classe Conexão responsável pela comunicação cliente servidor

```
private:
  void toServer();
  void shoot();
  void barrier();
  void checkCooldown();
  void createBullet();

  int side;
  int id;
  bool is_dead = false;
  bool barrier_on = false, can_barrier = true;
  bool can_shot = true, shot = true;

  int shield_cooldown = 0;
  int shot_cooldown = 0;

  const int max_shot_cooldown = 5;
  const int max_shield_cooldown = 15;
  const int shield_time = 5;
  const int player_movespeed = 3;
```

Ilustração 4.4 – Exemplo do arquivo-interface da classe Player.

dos futuramente, herdam da classe *GameObject*, que basicamente utiliza a classe *Position*, possibilitando a movimentação dos objetos pelo cenário.

No jogo, somente uma ação pode ser executada por vez, e ações que estiverem em tempo de recarga estarão desabilitadas. Propositalmente, estes testes são verificados no próprio cliente do jogo, então, modificando-o, o cliente pode burlar essas regras, e por exemplo, atirar tiros em um intervalo de tempo menor do que o correto, ou usar seu escudo mais frequentemente que seus adversários.

Todas ações executadas por um cliente são armazenadas em um arquivo, que por sua vez, são analisados com alguns dos métodos estudados no Capítulo 3.

Para facilitar a implementaçãos dos métodos e poder analisar somente seu tempo de processamento, utilizou-se um histórico de mensagens, contendo mensagens enviadas de um cliente ao servidor referente as suas ações executadas.

4.1 IMPLEMENTAÇÃO COM EXECUÇÃO SIMBÓLICA

Para implementar uma verificador simbólico, deve-se construir uma execução simbólica sobre a aplicação alvo. Visando facilitar os testes e a identificação do custo computacional das verificações das mensagens do cliente, elas foram armazenadas em um arquivo chamado *log*. Neste arquivo, estão armazenadas todas ações executadas por um cliente ao longo de *n* ciclos, onde *n* é o número de linhas do arquivo, e em cada linha dispoe-se a ação realizada. Para gerar o *log*, rodou-se a aplicação do cliente e do servidor durante vários minutos, fazendo com que o cliente faça ações aleatórias em cada um de seus ciclos, enquanto o servidor as armazena no arquivo *log*. Assim, construiu-se uma sequência de ações válidas no arquivo *log*, que foi lida pelo executável simbólico criado. Esta estratégia simula a validação das mensagens do cliente a partir de seu *log* gerado, ou seja, é uma verificação posterior às ações realizadas pelo jogador. Para melhorar o desempenho deste tipo de estratégia, pode-se rodar o algoritmo em apenas parte do *log* armazenado, ou quando se desconfia de um cliente específico após evidências negativas dele vindas de outros clientes.

O programa usado para gerar a execução simbólica de *Shooterman* foi KLEE, uma máquina virtual simbólica que é construída sobre a estrutura de um compilador LLVM, e disponível gratuitamente.(TEAM,) KLEE é baseado em um resolvedor de restrições STP, que implementa uma lógica para encontrar a solução dos conjuntos de restrições impostos a variáveis através da execução simbólica. Para facilitar sua instalação, o KLEE foi utilizado em uma imagem do Docker. Docker é uma plataforma *Open Source* que facilita a criação e administração de ambientes isolados.(INC,). A partir da imagem obtida do KLEE, foi possível criar um container, contendo o necessário para a utilização das ferramentas e bibliotecas disponíveis da máquina virtual simbólica.

Para o programa ser executado na máquina simbólica, ele deve primeiramente ser compilado como um arquivo *bitcode* no formato LLVM. Para isso utilizou-se o Clang++, com a *flag* -emit-llvm. Como vários arquivos são utilizados, usou-se o *llvm-link* para permitir que vários arquivos *bitcode* LLVM juntem-se em um único arquivo *bitcode*(PROJECT,). Infelizmente, apesar do KLEE reconhecer o arquivo *bitcode* gerado como válido, ele não reconheceu chamadas externas de funções e construtores de outras classes, retornando uma falha de chamada externa logo na construção da primeira classe instanciada.

Para prosseguir com a implementação, o código criado foi modificado para ser compatível em C. Para facilitar essa modificação, somente os arquivos mais essenciais do controle do fluxo do jogo foram convertidos, ocupando o único arquivo *client_main.cpp*. Como a biblioteca *JSON for Modern C++* não pôde ser utilizada, rodou-se um *script* em Python para converter os dados no formato *json* em um formato mais prático de ser lido em C. O A partir deste novo arquivo, pode-se ler linha por linha do arquivo, atribuindo os valores lidos para variáveis usando a função *sscanf* do C.

Na implementação, somente a variável *action* é representada simbolicamente. Nota-se que na Figura 4.5, a variável *action* representa uma ação que o jogador tomou. Como ela torna-se simbólica, ela pode representar qualquer valor, variando de 0 a 5, refletindo consequentemente uma das ações possíveis realizadas pelo jogador. A função *randomMovement()* desta implementação atualiza as informações do personagem de acordo com a ação realizada. Como a ação é simbólica, considera-se que todas ações possíveis do jogador foram testadas, e se uma delas resultou na equivalência com a ação registrada no log, a ação registrada no log é considerada semanticamente autêntica. O retorno dessa função *randomMovement()* retorna a validação da ação. Como cada valor possível de *action* cria um novo ramo de execução, o ramo deve ser finalizado se mostrar-se inválido semanticamente, por isso a função é retornada caso aquela ação seja considerada inválida. Os valores válidos, entretanto, continuam executando e ramificando novamente, na próxima iteração do *while*.

```
    while ((read = getline(&line, &len, file)) != -1){
        klee_make_symbolic(&action, sizeof(action), "action");
        if (randomMovement(action, line) == false)
            return;
    }
```

Ilustração 4.5 – Exemplo do código de execução simbólica usando KLEE.

4.2 IMPLEMENTAÇÃO DO SSIP

A partir da definição de estados abstratos e concretos do Capítulo 3.5.1, pode-se comparar as informações enviadas ao servidor de *Shooterman* como abstratas e as informações contidas no cliente como concretas. Levando em consideração, por exemplo, o escudo que o jogador pode ativar para tornar-se imune aos projéteis inimigos, o servidor somente tem acesso a informação da ativação do escudo, e não de seu tempo de recarga. Em jogos mais complexos, outras informações associadas a ações do jogador, como teste de colisões ao se movimentar também podem ser consideradas como informações concretas, mas que se fossem enviadas diretamente ao servidor e calculadas lá a todo ciclo do cliente ocasionariam em processamento extensivo demais para ser realizado pelo servidor para todos clientes.

Como a autenticidade dos usuários não foi levada em consideração, o uso de *hashs* e autenticadores de mensagens foi desconsiderado tanto na implementação quanto nos custos atribuídos a utilização deste método.

Analisando o SSIP, percebe-se que para realizar a verificação dos estados do cliente, o servidor de auditoria deve possuir acesso ao estado concreto inicial do cliente (referente ao Protocolo de Inicialização, Capítulo 3.5.4), acesso a todas *diffs* entre os estados concretos de t_a a t_0, assim como acesso a todas *diffs* entre os estados abstratos de t_a a t_0.

Modificando o cliente, foi criado um arquivo durante sua conexão com o servidor, que armazena informações dos estados concretos, *diffs* abstratas e *diffs* concretas de todos estados ao longo de uma partida. Este histórico de estados é então lido por um programa, que na prática atua como um servidor de auditoria. Após ler os conteúdos correspondentes aos estados e Δ's, os passos do protocolo de auditoria (mais detalhes no Capítulo 3.5.6) foram executados para verificar a integridade dos estados.

Primeiramente, a partir do estado concreto inicial, é calculado todos estados concretos a partir das *diffs* concretas obtidas. Como todos estados foram armazenados no formato *json*, foi relativamente fácil de atribuir as mudanças Δ aos estados. A figura 4.6 ilustra o processo do quarto passo do protocolo de auditoria do Capítulo 3.5.6.

A parte que demanda maior custo computacional deste protocolo é o quinto passo, onde é verificado se cada Δ concreto é válido ou não. Neste modelo, a partir de cada estado do jogo, é vericado se as ações executadas condizem com a integridade e as regras que a NVE deve cumprir. Ou seja, para cada mudança concreta ocorrida, é analisado se ela poderia ter ocorrido ou não, ou se outra mudança deveria ocorrer no lugar. Um exemplo simples disso seria da utilização da barreira. Caso ela tenha sido usada em um ciclo anterior, os próximos *diffs* concretos devem manter a atualização do tempo de recarga da barreira, até ela poder voltar a ser utilizada. Se o tempo de recarga não apareceu entre uma *diff* e outra após o cliente ter usado sua barreira, ou se a diferença do tempo de re-

```
        auto a = concrete_states.begin();

        next_states.push_back((*a));

        auto delta = concrete_delta.begin();
        string actions[] = {"position", "barrier", "can_barrier", "shield_cooldown",
        "shoot", "can_shot", "shot_cooldown"};

        for (delta; delta != concrete_delta.end(); delta++){
            json new_state;

            json delta_json = (*delta);

            for (string s : actions){
                if (delta_json.count(s) != 0){
                    new_state[s] = delta_json[s];
                }
            }

            next_states.push_back(new_state);
        }
```

Ilustração 4.6 – Exemplo de criação dos estados a partir do estato concreto e Δ concretos.

carga entre uma *diff* e outra é maior do que um, conclui-se que o jogador está escondendo informações ou modificando os dados no cliente de jogo.

5 RESULTADOS

Como abordado no Capítulo 4, os métodos de validação com execução simbólica e o SSIP foram implementados a partir do jogo *Shooterman* criado a fim de analisar estes métodos e estudar seus comportamentos e aplicações. Todos experimentos foram executados 10 vezes, de onde obteve-se uma média aritmética de seus tempos de execução. Ambos métodos foram avaliados utilizando uma estrutura de arquivo contendo a sequência de ações realizadas por um cliente, facilitando assim o acesso aos dados de tempo de execução do processamento de validação em si, desconsiderando-se as falha de pacotes, latência da rede e outras características que poderiam alterar o tempo total de execução do método.

A máquina usada para os experimentos possui um processador *quad-core* Intel(R) i5-2500 CPU, com 3.30GHz. O sistema operacional utilizado foi o Debian 4.6.2-2. A versão do compilador gcc utilizada foi a 6, e a do clang foi a 3.4.

Ambos métodos conseguiram encontrar trapaças em todos sequências de ações projetadas. Em situações onde, por exemplo, o usuário atira em um intervalo de tempo que deveria ser proibido pela regras da aplicação, ambos métodos conseguiram reconhecer a trapaça. Situações semelhantes aconteceram quando o usuário tentou ativar seu escudo em momentos que ele estava em tempo de recarga.

Para a obtenção do tempo de execução da verificação de execução simbólica, foi utilizado o comando *time* durante o período de execução do programa. Para a verificação com o sistema de auditoria, foi utilizado a biblioteca *chrono* do C++, possibilitando calcular somente o tempo de processamento das verificações, desconsiderando-se o tempo de leitura do arquivo com as ações verificadas. Esta estratégia foi adotada para ser mais fiel ao protocolo de auditoria, onde o processamento de verificação dos estados do cliente não acontece sobre arquivos armazenados no disco, mas sim sobre os dados já armazenados em memória.

5.1 RESULTADOS DA EXECUÇÃO SÍMBOLICA APLICADA A SHOOTERMAN

Percebe-se que apesar de ser uma implementação simples de execução simbólica, e dos ramos que surgiram a partir das restrições geradas terem um fluxo de operações limitados, o tempo de execução da validação aumenta consideravelmente de acordo com o número de validações a serem geradas. Isto provavelmente se deve ao fato de existir uma quantidade significativa de restrições existentes repetirem o mesmo fluxo de processamento executado pelas restrições anteriores. Ou seja, quando chega na terceira ação a ser validada, por exemplo, repete-se o processo de validação da primeira ação e da

segunda ação.

Para uma sequência pequena a ser validada de 100 ações, o tempo de execução para a comparação foi de 1.5 segundos, valor já considerado alto para uma pequena quantidade de dados. Conforme o número de ações a ser validado aumenta, como o Gráfico 5.1 ilustra, o tempo de validação aumenta drasticamente, principalmente após a marca de 600 ações, onde a validação chega a passar de três minutos. Pode-se concluir que quanto mais restrições geradas pela execução simbólica, maior se torna o custo para computar seus próximos ramos. Uma solução para otimizar o custo para resolver de uma sequência grande de restrições pode ser a alteração do algoritmo resolvedor de restrições STP, visando reduzir o número de execuções duplicadas dos ramos. Outra estratégia que pode ser adotada para melhorar o desempenho da execução simbólica na validação das ações de um cliente é a verificação de sequências menores de ações. Como o número de restrições acaba sendo menor, o tempo total acaba sendo bem menor comparado a sequências maiores, como as apresentadas neste trabalho.

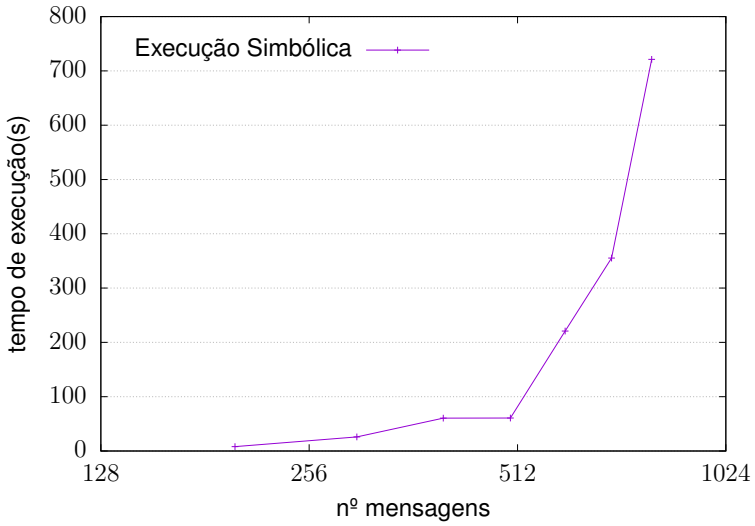

Gráfico 5.1 – Resultados obtidos com validação por execução simbólica.

5.2 RESULTADOS DO SSIP APLICADO A SHOOTERMAN

Os resultados do protolo de integridade semântica segura mostraram-se eficientes. Como o Gráfico demonstra, o método conseguiu verificar grandes quantidades de blocos de ações em um curto período de tempo. Para sequências de mensagens com tamanho inferior a 20000, o tempo de execução das verificações foi na escala de microsegundos, com 23 microsegundos de execução tanto para sequências de 10000 e 20000 mensagens. O tempo começou a aumentar em sequências maiores, mas ainda manteve-se sempre

próximo a faixa de tempo de um segundo. Percebe-se que a cada 10000 novas mensagens a serem processadas, o tempo de execução aumenta aproximadamente 0.15 segundos, mantendo-se em um crescimento linear.

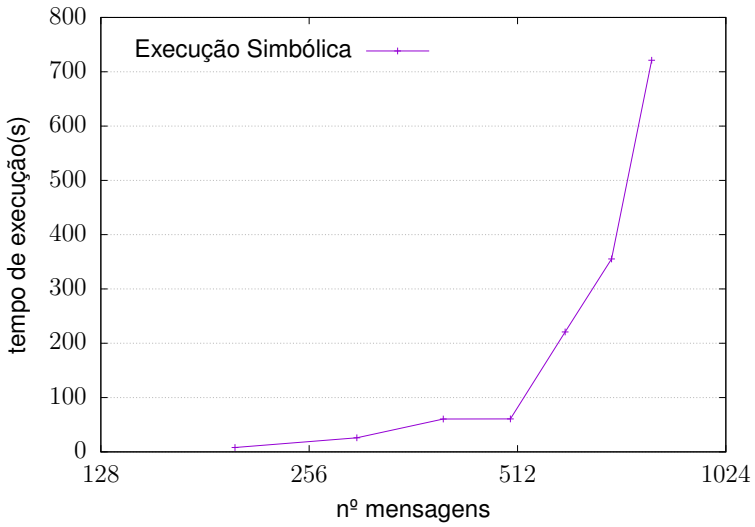

Gráfico 5.2 – Resultados obtidos com validação com auditoria.

Em um cenário convencional, dificilmente o servidor de auditoria precisaria avaliar sequências tão grandes de ações executadas pelo cliente. Mesmo assim, o algoritmo do protocolo mostrou-se eficiente para lidar com sequências de até mesmo 80000 ações. Comparando com os resultados obtidos da verificação de mensagens utilizando execução simbólica, o SSIP mostrou-se muito superior em questões de desempenho de tempo de execução. Além de mostrar-se eficiente na verificação das ações de *Shooterman*, o protocolo mostra-se versátil por dar a opção do desenvolvedor escolher de quantos em quantos ciclos de um cliente inicia-se um processo de auditoria. Além disso, como apenas uma auditoria é feita com um cliente por vez, o processamento gasto pelo servidor de auditoria acaba sendo bem reduzido.

Entretanto, algumas características negativas podem ser atribuídas ao SSIP. Considerando que, a cada ciclo do cliente ele envia informações adicionais contendo as mudanças concretas de seus estados ao servidor de auditoria, o número de mensagens a ser enviada dobra. Além de enviar as alterações abstratas ao servidor da aplicação, as alterações concretas (que possuem tamanhos maiores de mensagens) também são enviadas.

6 CONSIDERAÇÕES FINAIS

Esse projeto teve como propósito estudar diferentes estrátegias de detecção de trapaças em jogos *online*, focando no conjunto de trapaças que envolvem a adulteração do jogo, e analisar suas características.

Na etapa de desenvolvimento, foi projetado e implementado um pequeno jogo *online* em C++, no modelo cliente-servidor. Este jogo, chamado *Shooterman* foi alvo de dois métodos estudados que verificam as mensagens enviadas pelos clientes, encontrando possíveis usuários trapaceiros e atividades ilegais. Estes dois métodos foram implementados modificando o servidor utilizado no jogo, que verificava sequências de mensagens enviadas por clientes armazenadas em arquivos.

Ao final desta etapa, foi avaliado o desempenho dos métodos baseado no tempo que demoraram para verificar sequências de diferentes tamanhos de mensagens. Ambos métodos conseguiram encontrar sequências inválidas de mensagens enviadas pelo cliente, entretanto, o método que utiliza o protocolo SSIP mostrou-se superior em realizar esta tarefa. O verificador utilizando execução simbólica mostrou-se muito lento, principalmente quando o número de mensagens verificadas passou de 600, levando alguns minutos para realizar esta tarefa.

Como trabalhos futuros, novos métodos de verificação podem ser implementados e analisados com maior profundidade, assim como jogos maiores e complexos podem ser alvo dos métodos, para comparar seus desempenhos em infraestruturas maiores e modelos de ações mais complexos.

REFERÊNCIAS BIBLIOGRÁFICAS

BAUGHMAN, B. L. N. **Cheat-proof Playout for Centralized and Distributed Online Games**. [S.l.]: Proc. of the Twentieth IEEE INFOCOM Conference, 2001.

BETHEA, D.; COCHRAN, R. A.; REITER, M. K. Server-side verification of client behavior in online games. **ACM Trans. Inf. Syst. Secur.**, ACM, New York, NY, USA, v. 14, n. 4, p. 32:1–32:27, dez. 2008. ISSN 1094-9224. Disponível em: <http://doi.acm.org/10.1145/2043628.2043633>.

BLIZZARD. **StarCraft**. 2017. Acesso em 3 nov. 2017. Disponível em: <https://starcraft.com/pt-br/>.

COLLINGBOURNE, P.; CADAR, C.; KELLY, P. Symbolic crosschecking of data-parallel floating-point code. v. 40, p. 710–737, 07 2014.

GOOGLE Inc. 2017. Disponível em: <https://www.google.com.br/>.

GURU, G. G. **What is Go?** Acesso em 18 set. 2017. Disponível em: <https://gogameguru.com/what-is-go/>.

INC, D. **Docker**. Acesso em 19 nov. 2017. Disponível em: <https://www.docker.com/>.

JHA, S. et al. Enforcing semantic integrity on untrusted clients in networked virtual environments. In: **Proceedings of the 2007 IEEE Symposium on Security and Privacy**. Washington, DC, USA: IEEE Computer Society, 2007. (SP '07), p. 179–186. ISBN 0-7695-2848-1. Disponível em: <http://dx.doi.org/10.1109/SP.2007.16>.

LOHMANN, N. **JSON for Modern C++**. 2017. Acesso em 17 out. 2017. Disponível em: <https://nlohmann.github.io/json/>.

MAC. Acesso em 23 nov. 2017. Disponível em: <https://en.wikipedia.org/wiki/Message_authentication_code>.

MCCREARY, S.; CLAFFY, k. Trends in wide area IP traffic patterns - A view from Ames Internet Exchange. In: **ITC Specialist Seminar**. Monterey, CA: [s.n.], 2000.

MüLLER, M. Computer go. In: **Artificial Intelligence, Vol.134, No.1-2 (Special issue on Games, Computers and AI)**. [S.l.: s.n.], 2002. p. 145–179.

NEWZOO. **The Global Games Market Reaches $99.6 Billion In 2016, Mobiles Generating 37%**. 2016. Acesso em 18 set. 2017. Disponível em: <https://newzoo.com/insights/articles/global-games-market-reaches-99-6-billion-2016-mobile-generating-37/>.

PONG. 2017. Disponível em: <https://pt.wikipedia.org/wiki/Pong>.

PRADO, J. **Um computador do Google venceu um campeão mundial neste jogo chinês**. 2016. Disponível em: <https://tecnoblog.net/192604/computador-google-vence-campeao-go/>.

PROJECT, L. **LLVM Home | Documentation**. Acesso em 22 nov. 2017. Disponível em: <https://llvm.org/docs/index.html>.

REICHENBACH, C. **Foundations of Programming Languages**. 2015. Acesso em 23 nov. 2017.

SPOHN, D. **Cheating in Online Games**. Agosto 2016. Disponível em: <http://internetgames.about.com/od/gamingnews/a/cheating.htm>.

STEAM. **Steam**. 2003. Acesso em 20 set. 2017. Disponível em: <http://store.steampowered.com/?l=portuguese>.

TEAM, T. K. **KLEE LLVM Execution Engine**. Acesso em 17 nov. 2017. Disponível em: <http://klee.github.io/>.

YAN, J.; RANDELL, B. A systematic classification of cheating in online games. In: **Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games**. New York, NY, USA: ACM, 2005. (NetGames '05), p. 1–9. ISBN 1-59593-156-2. Disponível em: <http://doi.acm.org/10.1145/1103599.1103606>.