Saucy Express: viabilizando a geração automática de assinaturas para novas variantes de *malware*

Lucas França de Jesus[1], Guilherme Alt Merklein[1], Rafael Oliveira da Rocha[1 2], Lourenço Alves Pereira Júnior[1] e Idilio Drago[2]

[1]Instituto Tecnológico da Aeronáutica, São José dos Campos/SP - Brasil

[2]Università di Torino, Turim/TO - Itália

***Resumo*— A geração automática de assinaturas para novas variantes de *malware* atualmente enfrenta o desafio da ineficiência computacional devido à complexidade do problema. Além disso, o alto volume e a rápida evolução das ameaças sobrecarregam as equipes de defesa e tornam inviável depender apenas da análise manual, que pode levar horas. Isso cria janelas de vulnerabilidade, já que a maioria das soluções existentes são lentas demais para aplicação em tempo real ou enfrentam outras limitações como restrição de escopo. Como solução, este trabalho propõe o *Saucy Express*, uma arquitetura otimizada que implementa melhorias nas etapas extração de *features*, no cálculo da matriz de distâncias e na clusterização do *framework SAUCY SPICE*. Os resultados mostraram-se significativos, reduzindo o tempo médio de processamento de 1.551s para 15s, um speed-up de 98,8 vezes em comparação com a versão original, aumentando a sustentabilidade da solução em mais de 26 vezes no cenário analisado.**

***Palavras-Chave*— geração automática de assinaturas, detecção de malware, otimização de desempenho.**

I. Introdução

O cenário atual da cibersegurança é marcado por um volume crescente de tentativas de infecção por *malware*: apenas em 2024, as ferramentas de proteção da *Kaspersky* bloquearam mais de 300 milhões de ataques relacionados a *malware* e detectaram mais de 85 milhões de artefatos únicos [1]. A rápida evolução das ameaças e o emprego de técnicas como ofuscação, empacotamento e metamorfose dificulta suas detecções e sobrecarrega equipes de defesa [2], [3]. A análise de novas amostras, crucial para atualizar mecanismos de proteção, depende majoritariamente de analistas humanos e pode levar horas [4], [5]. Diante dessa realidade, torna-se clara a necessidade de se desenvolverem abordagens automatizadas capazes de bloquear eficientemente novas variantes, contribuindo assim para a redução da janela de vulnerabilidade entre a primeira detecção de uma nova ameaça e a ampla propagação, entre agentes de defesa, de regras de bloqueio eficazes para essa variante.

Uma das formas mais utilizadas atualmente para detectar *malwares* é baseada em assinatura [6], [2], entendida, no contexto deste trabalho, como a utilização de *features* (propriedades) de um binário para identificá-lo. Enquanto propriedades simples, como *hashes*, são rapidamente extraídas e não possuem capacidade de generalização (isto é, de representar diversas variantes), *features* como chamadas de sistema podem ser capazes de representar um conjunto de cepas, mas exigem intenso esforço manual no processo de análise de *malware*.

Com o objetivo de reduzir a necessidade da intervenção humana, diversos trabalhos anteriores propuseram métodos automatizados para a geração de assinaturas. Algumas abordagens exploram o próprio mecanismo metamórfico do *malware* para produzir automaticamente assinaturas de suas variantes [7], [8], [9]. Outras soluções aplicam técnicas avançadas de inteligência artificial — como *Large Language Models* (LLMs) e *Deep Learning* — para inferir regras de detecção [10], [11]. Entretanto, todas essas soluções ainda enfrentam limitações significativas como restrições de escopo de aplicação, baixa explicabilidade do resultado ou incapacidade de generalizar para novas variantes de *malware*.

Como alternativa a essas abordagens, um trabalho recente, chamado *SAUCY SPICE*, propôs um método de clusterização não supervisionado para agrupar e extrair características comuns, gerando automaticamente assinaturas capazes de representar grandes conjuntos de variantes de *malware* [12]. Além disso, a solução é completamente flexível e fornece informações interpretáveis sobre as assinaturas geradas. Entretanto, em seu desenho original, a solução sofre com sérias restrições de eficiência computacional, o que limita sua adoção em cenários de análise em escala e em tempo real.

Buscando abordar essas limitações, o presente trabalho introduz o *Saucy Express*: uma extensão do *SAUCY SPICE* otimizada para geração automática de assinaturas com alto desempenho. Para isso, executamos uma avaliação minuciosa da eficiência computacional no módulo de assinaturas do *SAUCY SPICE* e implementamos diversas otimizações.

Como principais contribuições deste trabalho, destacam-se:

1) a proposta e implementação do *Saucy Express*, um módulo de geração automática de assinaturas eficiente e flexível;
2) otimizações computacionais que resultam em um *speed-up* de até 98,8×;
3) avaliação preliminar do impacto da utilização de um novo tipo de *feature* na eficácia e eficiência do modelo;

O restante deste artigo é estruturado como descrito a seguir. A Seção II descreve em maiores detalhes outros trabalhos sobre o tema e faz uma análise comparativa entre eles e este estudo. A Seção III ambienta o leitor com a estrutura e funcionamento do *SAUCY SPICE*, enquanto a Seção IV descreve as otimizações propostas por nossa solução. A Seção V detalha como foram realizados os experimentos deste trabalho, expõe e discute os resultados obtidos. Por fim, a Seção VI traça algumas conclusões e propõe direcionamentos para trabalhos futuros.

L. F. Jesus, dejesus@ita.br; G. A. Merklein, galtmerklein@gmail.com; R. O. Rocha, rafaelror@ita.br; L. A. P. Júnior, ljr@ita.br; I. Drago, idilio.drago@unito.it.

II. Trabalhos Relacionados

Diversos estudos anteriores abordaram a geração automática de assinaturas a partir de amostras conhecidas, mas cada um com escopo limitado. Alguns concentraram-se em detectar componentes específicos do binário malicioso, como *packers* [13] ou funções individuais [14]. Embora úteis como suporte à análise, essas abordagens não resolvem o problema geral da criação de regras para *malware* em sua totalidade.

Outro grupo de trabalhos dedicou-se a *malwares* polimórficos e metamórficos, aproveitando seus próprios mecanismos de mutação para gerar assinaturas para todas suas variações [7], [8], [9]. Apesar de robustos contra mutações, esses métodos exigem conhecimento prévio do processo de ofuscação utilizado, geram grande quantidade de assinaturas e não se aplicam a famílias de *malware* fora desse perfil.

Um terceiro conjunto de soluções buscou maior cobertura de espécies de *malware*, mas enfrenta restrições práticas: algumas dependem de estruturas internas de sistemas operacionais específicos [15]; outras requerem *features* dinâmicas obtidas via *sandbox* [11] ou mesmo o código-fonte do *malware* [10], nem sempre disponíveis no contexto real de análise.

Mais recentemente, um trabalho [12] foi além e apresentou um módulo de geração automática de assinaturas genéricas flexível e eficaz integrado com um módulo de detecção em tempo real executado totalmente em *kernel mode*, possibilitando grande rapidez no bloqueio de ameaças sem *overheads* significativos. No entanto, pelo seu escopo mais abrangente, seu enfoque de eficiência se limitou ao último módulo apenas, sendo a geração de assinaturas inviável para pronta-resposta.

Nossa solução, *Saucy Express*, utiliza como base o módulo de geração de assinaturas do *SAUCY SPICE*, já validado em termos de acurácia e capacidade de generalização, para focar exclusivamente na sua eficiência computacional. Com isso, desenvolvemos uma solução eficiente que atua estaticamente em todo tipo de binário para produzir uma assinatura completa para qualquer classe de *malware*.

III. Conceitos Fundamentais

Esta seção resume os conceitos necessários para compreender as otimizações apresentadas na Seção IV.

A. Pipeline original do SAUCY SPICE

O SAUCY SPICE foi proposto como uma solução **automática** de geração de assinaturas YARA que agrupa amostras benignas e maliciosas para extrair padrões generalistas. O processo *offline* de criação de políticas compreende seis passos principais:

1) **Extração de *features***: para cada executável PE (*Portable Executable*), o framework exporta um documento de texto contendo todas as suas chamadas API, uma por linha, no formato "(*library*) (*function*)". Cada executável é totalmente mapeado em memória antes de acessar a IAT (*Import Address Table*).
2) **Preparação de amostras**: cada arquivo resultante do passo anterior é transformado em um vetor binário $\mathbf{f}_i \in \{0,1\}^d$, em que d é o tamanho do vocabulário de *imports* coletado no *dataset* de treino, indicando a presença de cada função importada. Esses vetores servem de base para a fase seguinte.
3) **Construção da matriz de distâncias $\mathbf{D}$**: utiliza-se a *similaridade do cosseno* entre os vetores $\mathbf{f}_i$ e $\mathbf{f}_j$. O elemento D_{ij} é obtido por (1).

$$D_{ij} = 1 - \frac{\mathbf{f}_i^\top \mathbf{f}_j}{\|\mathbf{f}_i\| \cdot \|\mathbf{f}_j\|},\ 1 \le i < j \le n \qquad (1)$$

   A matriz triangular superior de dimensões $n(n-1)/2$ é calculada *do zero* a cada execução.
4) **Clusterização via DAMICORE**: o módulo DAMICORE [16] recebe a matriz de distância $\mathbf{D}$ e produz uma partição $\mathcal{C} = C_1, \ldots, C_k$ das amostras, maximizando modularidade por meio dos algoritmos *Neighbor-Joining* e *FastGreedy*.
5) **Seleção de parâmetro**: são testados classificadores com base em diferentes ϵ, parâmetro que define a proporção de amostras maliciosas necessárias em um *cluster* para que ele seja usado para criar assinaturas.
6) **Geração de assinaturas**: são extraídas as chamadas *APIs* presentes em cada *cluster* predominantemente malicioso ($r_M > \epsilon$ no classificador que obteve maior precisão) e a partir delas são geradas regras YARA.

B. Métrica vetorial de similaridade

Apesar de o DAMICORE original utilizar a *Normalized Compression Distance* (NCD) como método para cálculo da matriz de distância, Chahud et al. [12] relataram, já no artigo inicial, a adoção da **similaridade cosseno** no módulo de clusterização do SAUCY SPICE por apresentar melhores resultados. Esta última métrica apresenta complexidade linear em relação à dimensionalidade do vetor, $\mathcal{O}(d)$ por par de amostras, e consequentemente $\mathcal{O}(n^2 d)$ para cálculo de uma matriz de distância entre n vetores.

C. Gargalos de desempenho identificados

Nos testes preliminares e análise do código de referência, utilizado no artigo original, o principal gargalo identificado foi o cálculo da matriz $\mathbf{D}$ devido à avaliação exaustiva dos $\binom{n}{2}$ pares a cada execução e *overhead* causado pela paralelização implementada. As etapas de clusterização e de extração de *features*, apesar de muito menos problemáticas nessa análise inicial, também apresentaram indícios de que cresceriam rapidamente conforme o incremento da quantidade de amostras utilizadas, em especial a fase de execução do algoritmo *Neighbor-Joining*, que utilizava sua implementação clássica de complexidade teórica $\mathcal{O}(n^3)$. Estes achados serviram de motivação direta para as otimizações propostas na próxima seção.

D. Features

A implementação original do SAUCY SPICE utiliza como entrada um vetor binário que indica a presença de chamadas de sistema na amostra. Embora esse tipo de *feature* seja comprovadamente eficaz para representar arquivos executáveis, a arquitetura do SAUCY SPICE é flexível e pode acomodar outros formatos de entrada. Trabalhos anteriores demonstraram que abstrações de chamadas de sistema — por exemplo, agrupar as chamadas por família em vez de tratá-las individualmente — capturam melhor a semântica das *syscalls* e permitem representar com uma mesma *feature* um conjunto

mais amplo de amostras semanticamente semelhantes [17], [18]. Assim, modificações no formato das *features* originais do SAUCY SPICE têm potencial para melhorar tanto a eficácia quanto a eficiência do modelo. Essa hipótese é investigada de forma preliminar nos experimentos apresentados neste trabalho.

IV. Arquitetura Saucy Express

A Figura 1 apresenta a arquitetura do Saucy Express, uma versão otimizada do módulo de geração de assinaturas do SAUCY SPICE que incorpora quatro melhorias principais, detalhadas a seguir, sem alterar a lógica central do algoritmo original.

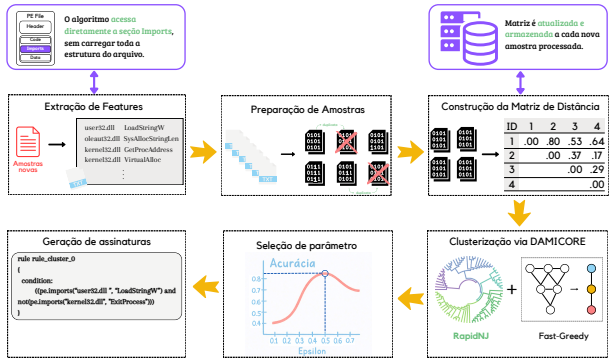

Fig. 1
Arquitetura do Saucy Express.

a) Otimização da matriz de distâncias: O cálculo da matriz de distâncias entre amostras era o principal gargalo do módulo original, com alto custo de *overhead* pela forma que a paralelização estava implementada e complexidade $\mathcal{O}(n^2d)$ para n amostras conhecidas e d chamadas de API. No Saucy Express, além de corrigirmos o código de paralelismo, armazenamos persistentemente as distâncias calculadas entre as amostras já processadas. Assim, a cada nova execução, apenas computamos distâncias envolvendo as amostras recém-chegadas (quantidade k), resultando em uma complexidade reduzida de $\mathcal{O}(d(nk+k^2))$. Como, em larga escala, o número de amostras que chega (k) é muito menor que o total de amostras já conhecidas (n), essa etapa se aproxima da complexidade linear (caso $d << n$) ou quadrática (caso d seja da mesma ordem de grandeza que n), melhorando acentuadamente o tempo de resposta em relação à arquitetura original.

b) Deduplicação de vetores de features: O fato de a base de conhecimento de *malware* ser cumulativa faz com que, em dado momento, a quantidade de amostras conhecidas passe a ser proibitiva para a geração de assinaturas. Entretanto, como as chamadas *API* são baseadas exclusivamente na análise estática da *IAT*, amostras semelhantes tendem a ter seus vetores de representação idênticos. Esses vetores repetidos aumentam o custo de processamento sem proporcionar nenhum ganho real para a qualidade do resultado. Por isso implementamos uma etapa de deduplicação na qual os vetores de importação (sequência binária) da base de amostras conhecidas são comparados e as duplicidades são descartadas, reduzindo o tamanho efetivo do conjunto de dados. Todo esse processo é feito de maneira *offline*, antes de iniciar o processo de geração de assinaturas.

c) Paralelização e parsing seletivo do PE: Nossa terceira modificação no pipeline focou em otimizar a eficiência da análise do binário original. Para isso, aplicamos paralelização na etapa de extração de *features* e modificamos a forma como o Portable Executable (*PE*) é importado. Em vez de analisar o cabeçalho completo, agora o processo se concentra apenas na Import Address Table (*IAT*).

d) Otimização da etapa de Neighbor-Joining: O algoritmo utilizado para gerar a árvore filogenética das amostras possui sempre complexidade teórica $\mathcal{O}(n^3)$ em sua forma canônica. Para reduzir seu impacto no tempo de geração de assinaturas, implementamos o algoritmo *Rapid Neighbor-Joining* (RapidNJ) [19] que, no geral, tem demonstrado reduzir a complexidade para $\mathcal{O}(n^2)$ utilizando uma técnica otimizada de busca pelos pares de nós a serem agrupados sem alterar o critério de decisão original do algoritmo clássico. Isso garante o mesmo resultado da implementação anterior, pois no pior dos cenários, o RapidNJ apenas precisará percorrer todos os nós (retornando à complexidade cúbica).

Além de realizar essas implementações, avaliamos de forma preliminar a utilização de *features* baseadas em categorias de chamadas de *API*. A hipótese investigada foi a de que tal mudança permitiria melhorar a eficiência da solução, uma vez que a redução dimensional (de milhares de chamadas de API para algumas dezenas ou centenas de categorias de API) diminui o custo de processamento do cálculo da matriz de distância ao reduzir o componente linear d da complexidade $\mathcal{O}(n^2d)$ da etapa e tende a aumentar a taxa de duplicação de vetores ao agrupar funções semelhantes sob uma mesma *label*.

V. Experimentos e Resultados

Esta seção está dividida em três partes: i) impacto das modificações no pipeline; ii) efetividade da solução; e iii) impacto do novo conjunto de *features*. Todas as medições de tempo são em segundos e foram utilizadas amostras extraídas das mesmas fontes que o artigo original: os *goodwares* são binários do Windows Server 2022 e os *malwares* são amostras obtidas do site MalwareBazaar[1]. No entanto, foi utilizado apenas um subconjunto reduzido do *dataset* original devido a limitações de tempo para execução dos experimentos. Por fim, o fracionamento treino-teste manteve a proporção de 90% para aprendizado de regras e 10% para validação, idêntico ao do artigo original.

A plataforma de testes na qual foram executados os experimentos foi uma máquina virtual provisionada com 20 vCPUs (2,10 GHz), 62 GB de RAM e SSD de 1 TB. Não foi utilizada GPU. Seu sistema operacional era o Debian 11 “bullseye” (kernel 5.10.0-14-amd64) com Python 3.9.2 (CPython) em ambiente global. As versões das principais ferramentas utilizadas foram: NumPy 1.23.2, pandas 2.3.0, SciPy 1.13.1, scikit-learn 1.6.1, igraph 0.11.8, ETE3 3.1.3 e rapidNJ 2.3.3.

A. Impacto das modificações da pipeline

Foram desenvolvidos cinco pipelines para testes, cada um com as suas melhorias implementadas cumulativamente ao anterior, conforme descrito abaixo:

[1]https://bazaar.abuse.ch/

- **P0 — Baseline**: arquitetura original, com cálculo completo da matriz em toda rodada.
- **P1 — Matriz otimizada**: matriz de distância é melhor paralelizada e passa a ser atualizada de forma incremental com as novas amostras ao invés de ser recalculada do zero.
- **P2 — Deduplicação**: implementa a remoção de vetores binários duplicados.
- **P3 — Extração de *features* otimizada**: restringe leitura do *PE* à *IAT* e implementa paralelização na extração.
- **P4 — NJ otimizado:** implementa o algoritmo *RapidNJ* no lugar do algoritmo *Neighbor-Joining* canônico.

Para avaliar o impacto das modificações, foi medido o desempenho das pipelines para diversas quantidades de amostras. Iniciou-se o teste com 300 *goodwares* e 1 *malware*, em seguida foram adicionados 100 novos *malwares* a cada rodada de testes. No total, foram gerados 75 eventos para cada pipeline, sendo 5 repetições para cada n quantidades de amostras extras de *malwares*, que variou de 0 até 1.400.

Foram monitorados os tempos das seguintes etapas principais: extração de *features* (T_{feat}), construção/atualização da matriz de distância (T_{dist}), clusterização (T_{clust}) e total do módulo de geração de assinaturas (T_{tot} = T_{feat} + T_{dist} + T_{clust} + etapas administrativas), em que "etapas administrativas" agrega preparação de amostras, busca do melhor ϵ e geração das regras.

A Tabela I resume o tempo médio dos 75 ensaios por pipeline (média amostral ± desvio-padrão das observações). A coluna "$\times P0$" representa o fator de aceleração obtido por cada pipeline em comparação direta com o *baseline* P0 — ou seja, quantas vezes ele é mais rápido que a arquitetura original.

TABELA I
TEMPOS MÉDIOS ± DESVIO-PADRÃO (75 EXECUÇÕES).

Pipeline	$T_{\text{feat}}(s)$	$T_{\text{dist}}(s)$	$T_{\text{clust}}(s)$	$T_{\text{tot}}(s)$	× P0
P0	76,3 ± 26,0	1216 ± 899	249,6 ± 253	1551,7 ± 1 180	1,0
P1	76,7 ± 26,2	2,0 ± 1,1	250,3 ± 253	338,7 ± 283	4,6
P2	77,5 ± 27,8	0,7 ± 0,2	20,8 ± 13,0	110,7 ± 48	14,0
P3	0,8 ± 0,13	0,7 ± 0,2	20,5 ± 12	33,4 ± 19	46,4
P4	0,9 ± 0,20	0,87 ± 0,3	4,83 ± 1,85	15,71 ± 6,51	98,8

Para os pipelines 2, 3 e 4, que possuem deduplicação de amostras, uma média de aproximadamente 506 (± 317) vetores repetidos foram descartados nos experimentos, de maneira que apenas uma média de 460 (± 119) vetores únicos prosseguiram para as etapas de cálculo/atualização da matriz de distância e posteriores.

A Figura 2 evidencia a mudança progressiva de gargalo ao longo dos experimentos:

Analisando-se os tempos de cada pipeline, podemos observar que:

- **P1** praticamente elimina o custo de T_{dist} com uma redução média no tempo de execução de aproximadamente 618 vezes. Isto confirma a eficácia da otimização realizada no cálculo da matriz de distância.
- **P2** ataca o impacto do número de vetores em T_{clust}, reduzindo a quantidade de amostras efetivamente processadas pelo DAMICORE para uma média de 460, o que gera um ganho de 12 vezes no tempo da etapa. O ganho global, neste ponto, chega a 14 vezes.

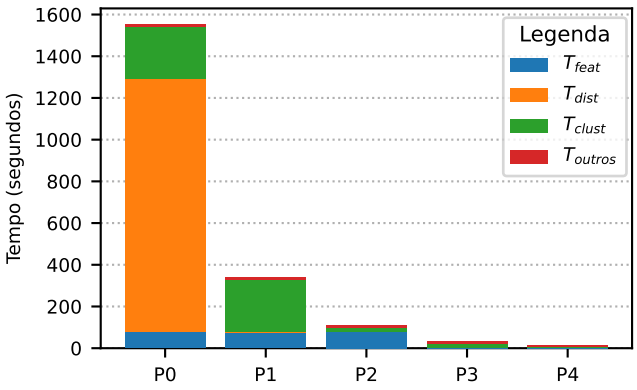

Fig. 2
DECOMPOSIÇÃO DO TEMPO TOTAL POR PIPELINE.

- **P3** otimiza a etapa de extração de *features*. A leitura seletiva da *IAT* combinada à paralelização faz T_{feat} despencar, na média, de 77 s para 0,78 s. Com isso, T_{clust} volta a responder por aproximadamente 60% do tempo de execução do pipeline.
- **P4** soluciona o último gargalo significativo restante, que é a complexidade cúbica do algoritmo *Neighbor-Joining*, gerando uma redução média de 73,90% no tempo de execução da etapa de clusterização.

Em síntese, as quatro intervenções reduziram o tempo total médio de execução de 1.551,75 s para 15,71 s, representando um *speed-up* de 98,8 vezes sem alterar a lógica do módulo de geração de assinaturas.

B. Efetividade da solução

Para avaliar a efetividade do Saucy Express, analisou-se o tempo para geração de assinaturas em cada etapa descrita na subseção anterior. A Figura 3 mostra uma comparação entre os tempos médios de execução de cada pipeline para cada quantidade de *malware* extras adicionadas no experimento. Nela é possível observar visualmente como o tempo prático de execução reduz de uma curva exponencial (*pipeline* 0, em azul) para, praticamente, uma reta — *pipelines* 2 (em verde), 3 (em vermelho) e 4 (em roxo).

Na prática, consideramos que uma solução é efetiva se seu tempo de resposta for inferior ao de um analista humano. Como essa métrica é bastante variável (dependendo de fatores como a capacidade técnica do analista, recursos disponíveis e técnicas de ofuscação empregadas no *malware*), adotamos o valor hipotético de 1 hora para comparação. Usando regressão polinomial sobre os dados de desempenho, estimamos para cada pipeline o número máximo de amostras que ainda cabe analisar em até 1 hora. Esses resultados estão na coluna "# amostras" da Tabela II.

Para prever por quanto tempo a solução permanecerá efetiva, é necessário saber a quantidade diária de amostras maliciosas analisadas. Entretanto, esses valores variam bastante com fatores que vão desde a capacidade de atacantes gerarem variantes até a qualidade de representação das assinaturas utilizadas. Apenas para fins de comparação, projetamos três cenários de surgimento diário de variantes que necessitam de novas assinaturas:

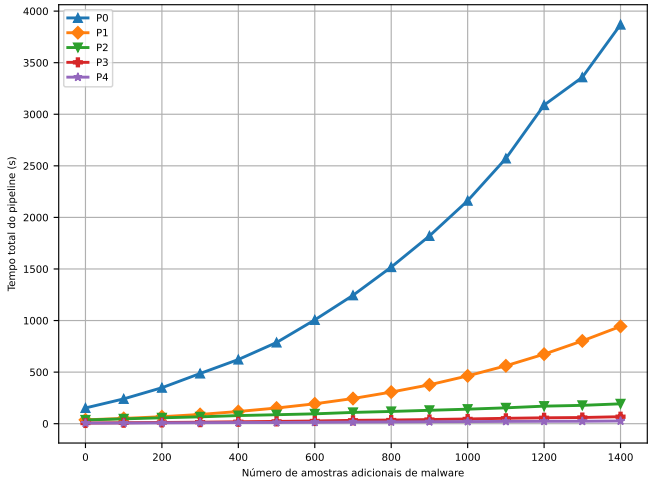

Fig. 3
COMPARAÇÃO DO TEMPO TOTAL DE CADA PIPELINE POR NÚMERO DE AMOSTRAS ADICIONAIS DE MALWARE.

- **Cenário 1 (leve):** 2 amostras/dia
- **Cenário 2 (moderado):** 10 amostras/dia
- **Cenário 3 (pesado):** 50 amostras/dia

A Tabela II indica em quantos dias cada pipeline deixa de ser efetivo (ou seja, ultrapassa 1 hora de processamento).

TABELA II
ANÁLISE DE EFETIVIDADE. UNIDADE DOS CENÁRIOS EM DIAS.

pipeline	# amostras	cenário 1	cenário 2	cenário 3
0	1608	804	161	32
1	3006	1503	301	60
2	13414	6707	1341	268
3	16675	8337	1668	333
4	42956	21478	4296	859

A partir dos dados hipotéticos é possível observar como a efetividade das soluções podem variar de poucos dias até meio século, a depender do cenário enfrentado. Apenas para comparação mais objetiva, definimos a *sustentabilidade relativa* de cada *pipeline* em relação à *pipeline* 0 dividindo o número de amostras suportado por pipeline i pelo valor correspondente da *pipeline* 0. Assim, para a coluna "# amostras":

$$\text{sustentabilidade}_i = \frac{\#\text{amostras}_i}{\#\text{amostras}_0}.$$

Obtivemos:

- *Pipeline 1:* $\approx 1{,}87\times$ mais sustentável que P0;
- *Pipeline 2:* $\approx 8{,}34\times$ mais sustentável que P0;
- *Pipeline 3:* $\approx 10{,}37\times$ mais sustentável que P0;
- *Pipeline 4:* $\approx 26{,}71\times$ mais sustentável que P0.

Ou seja, a solução final (P4) tem uma capacidade de processar um volume de ameaças acumuladas mais de **26 vezes maior** que a arquitetura original antes de atingir o mesmo teto de custo computacional. Além disso, os dados da tabela revelam o impacto das otimizações. Enquanto o pipeline original suportaria por apenas um mês (32 dias) o cenário mais pesado ou dois anos (804 dias) o cenário mais leve, as projeções para o *Saucy Express* (P4) sugerem uma viabilidade superior a dois anos (859 dias) e a 58 anos (21.478 dias) sob as mesmas condições.

C. *Impacto do novo conjunto de* features

Para analisar a hipótese relacionada à utilização de *features* semânticas, adotamos uma abordagem inicial e pragmática: o modelo gpt-4o-mini da *OpenAI* foi utilizado para gerar sinteticamente os nomes das *labels* e atribuí-las a chamadas de *API* do Sistema Operacional *Windows* a partir de *prompts* direcionados. A partir disso, um experimento comparativo foi conduzido com o *pipeline* P3 utilizando 5400 amostras, uma vez com as *features* originais (chamadas de API individuais) e outra com as novas *features* baseadas em *labels*. Apesar de ser um método experimental inicial, ratificamos que o objetivo principal desse procedimento foi apenas verificar, em caráter preliminar, o impacto de *labels* na taxa de duplicação de vetores e na acurácia da solução a fim de avaliar sua relevância para futuros estudos.

No total, foram geradas 416 *labels* distintas capazes de abarcar 19.430 chamadas de *API* únicas. Para efeito de comparação, a fim de se calcular a matriz de distância em um *dataset* de apenas 1.000 amostras que contemplasse todas essas chamadas, seria necessária a realização de $\approx 9,7 \times 10^9$ iterações apenas para cômputos dos produtos internos entre os vetores no caso da *feature* ser a chamada *API* individual, enquanto esse número cairia para $\approx 2,1 \times 10^8$ no caso de utilização das *labels*.

Nessa abordagem inicial, cada chamada pôde receber uma ou mais labels (por exemplo, à chamada API `CryptEncrypt` foram atribuídas as *labels* `crypto` e `write`). A Tabela III apresenta as 45 *labels* com frequência relativa igual ou superior a 0,1%.

TABELA III
LABELS CRIADOS POR IA GENERATIVA.

Label (Frequência Relativa)			
process (18.57%)	debug (1.28%)	copy (0.43%)	float (0.17%)
read (12.10%)	init (1.26%)	check (0.42%)	exception (0.16%)
write (10.41%)	security (1.05%)	cleanup (0.34%)	resource (0.14%)
file (7.12%)	convert (0.82%)	query (0.33%)	database (0.14%)
gui (6.75%)	math (0.79%)	search (0.26%)	io (0.12%)
network (5.80%)	data (0.63%)	control (0.25%)	sync (0.11%)
thread (4.44%)	compare (0.49%)	graphics (0.23%)	audio (0.11%)
string (4.32%)	input (0.47%)	error (0.23%)	timer (0.11%)
ipc (3.62%)	device (0.47%)	event (0.21%)	info (0.11%)
crypto (2.98%)	system (0.47%)	create (0.20%)	
memory (2.90%)	time (0.46%)	config (0.20%)	
registry (2.79%)	conversion (0.43%)	locale (0.18%)	

O melhor desempenho obtido em cada configuração nas execuções do experimento foi selecionado para comparação. A Tabela IV sintetiza os resultados:

TABELA IV
COMPARAÇÃO ENTRE OS ESQUEMAS DE FEATURES ORIGINAL E NOVO

Feature	Acurácia	Amostras Duplicadas
Original	0.97674	2239
Baseado em Labels	0.85546	2420
Variação	**-12,13 p.p.**	**+8,08%**

Conforme observado, o novo esquema de *features* apresentou uma redução de 12,13 pontos percentuais na acurácia. A queda de desempenho sugere que o vetor binário com as *labels* não capturou bem a semântica dos executáveis. No entanto, testes realizados sem a remoção de duplicatas atingiram uma

acurácia de 90.30%, o que consideramos um bom resultado considerando a origem das *features* novas.

Quanto à duplicidade, observamos um aumento de 8,08% na taxa de amostras duplicadas. Esse achado reforça o potencial da adoção de *labels* em impactar positivamente a eficiência do *pipeline*, uma vez que tende a gerar mais duplicatas por agrupar diferentes chamadas *API* em uma única *label* ao mesmo tempo em que reduz dimensionalidade dos vetores, contribuindo assim para geração de assinaturas ainda mais eficientes por meio de uma abordagem semelhante à utilizada atualmente.

Os resultados, portanto, indicam que a abordagem é promissora, mas requer um refinamento na criação das *labels*. A utilização de categorias mais específicas para a identificação de comportamentos maliciosos, em vez de genéricas, tem o potencial de recuperar a acurácia perdida e, ao mesmo tempo, ampliar os ganhos de eficiência.

VI. Conclusão

Com o propósito de transpor as barreiras de performance do *framework SAUCY SPICE*, este trabalho se dedicou a desenvolver e validar o *Saucy Express*, uma arquitetura voltada à geração automática de assinaturas de *malware*. Por meio de uma série de otimizações aplicadas nos principais gargalos identificados, como no cálculo da matriz de distâncias e na clusterização, buscou-se tornar a ferramenta viável para cenários realistas, nos quais a agilidade na resposta a ameaças é fundamental para diminuir janelas de vulnerabilidade. Os experimentos revelaram uma expressiva diminuição no tempo de processamento, o que aponta para um potencial de contribuição na agilidade com que os mecanismos de defesa podem ser atualizados frente a novas variantes de *malware*. A contribuição técnica central do projeto é um pipeline modificado, que alcançou um speed-up de até 98,8 vezes em relação à arquitetura original sem, no entanto, alterar a lógica que fundamenta o método.

Embora os avanços sejam significativos, o trabalho ainda apresenta algumas limitações. A análise de eficiência, por exemplo, foi conduzida sobre um conjunto de dados relativamente enxuto, em consequência de restrições de tempo. Idealmente, portanto, trabalhos futuros devem avaliar o *Saucy Express* em conjuntos de amostras maiores, mais heterogêneos e mais representativos, que cubram um maior *range* temporal. Outro ponto a se notar é que, apesar da otimização, a complexidade computacional da etapa de clusterização ainda se mantém quadrática — e, no pior caso, cúbica. Essa característica pode se tornar um novo gargalo, a depender de fatores como a frequência com que novos *malwares* surgem em uma determinada organização. Adicionalmente, a exploração preliminar de *features* semânticas, embora tenha indicado uma queda na acurácia, revelou um potencial para ampliar a taxa de deduplicação de amostras, sinalizando um caminho de pesquisa promissor que, contudo, ainda demanda um refinamento cuidadoso para harmonizar eficiência e eficácia.

Referências

[1] Kaspersky, "Kaspersky Security Bulletin 2024. Statistics — securelist.com," https://securelist.com/ksb-2024-statistics/114795/, 2024, [Acessado em 22-Jun-2025].

[2] A. Abusitta, M. Q. Li, and B. C. Fung, "Malware classification and composition analysis: A survey of recent developments," *Journal of Information Security and Applications*, vol. 59, p. 102828, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212621000648

[3] Ö. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, 2020.

[4] E. Downing, Y. Mirsky, K. Park, and W. Lee, "DeepReflect: Discovering malicious functionality through binary reconstruction," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3469–3486. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/downing

[5] K. Baker, "Malware Analysis: Steps & Examples — CrowdStrike — crowdstrike.com," https://www.crowdstrike.com/en-us/cybersecurity-101/malware/malware-analysis/, 2023, [Accessed 06-11-2024].

[6] C. Group, "2025 cyberthreat defense report," https://cyberedgegroup.com/cdr/, 2025, [Acessado em 22-Jun-2025].

[7] M. Campion, M. Dalla Preda, and R. Giacobazzi, "Learning metamorphic malware signatures from samples," *Journal of Computer Virology and Hacking Techniques*, vol. 17, no. 3, pp. 167–183, Sep 2021. [Online]. Available: https://doi.org/10.1007/s11416-021-00377-z

[8] H. Razeghi Borojerdi and M. Abadi, "Malhunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection," in *ICCKE 2013*, 2013.

[9] Y. Tang, B. Xiao, and X. Lu, "Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms," *Computers & Security*, vol. 28, no. 8, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404809000583

[10] X. Zhang, H. Chen, Y. He, W. Niu, and Q. Li, "Automatically generating rules of malicious software packages via large language model," 2025. [Online]. Available: https://arxiv.org/abs/2504.17198

[11] O. E. David and N. S. Netanyahu, "Deepsign: Deep learning for automatic malware signature generation and classification," in *2015 International Joint Conference on Neural Networks (IJCNN)*, 2015, pp. 1–8.

[12] L. Chahud, J. Favoretti, R. Rocha, N. S. Jr., F. Verri, I. Drago, and L. P. Junior, "Saucy spice: uma nova abordagem eficiente para a detecção de malwares baseada em assinatura," in *Anais do XXIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*. Porto Alegre, RS, Brasil: SBC, 2023, pp. 209–222. [Online]. Available: https://sol.sbc.org.br/index.php/sbseg/article/view/27208

[13] S. Li, J. Ming, P. Qiu, Q. Chen, L. Liu, H. Bao, Q. Wang, and C. Jia, "Packgenome: Automatically generating robust yara rules for accurate malware packer detection," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3078–3092. [Online]. Available: https://doi.org/10.1145/3576915.3616625

[14] J. A. Onieva, P. P. Jiménez, and J. López, "Malware similarity and a new fuzzy hash: Compound code block hash (ccbhash)," *Computers & Security*, vol. 142, p. 103856, 2024.

[15] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," in *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, San Diego, CA, February 2017.

[16] A. Sanches, J. M. Cardoso, and A. C. Delbem, "Identifying merge-beneficial software kernels for hardware implementation," in *2011 International Conference on Reconfigurable Computing and FPGAs*, 2011, pp. 74–79.

[17] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Trans. Priv. Secur.*, vol. 22, no. 2, apr 2019. [Online]. Available: https://doi.org/10.1145/3313391

[18] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, 2020, pp. 757–770.

[19] M. Simonsen, T. Mailund, and C. N. Pedersen, "Rapid neighbour-joining," in *Algorithms in Bioinformatics: 8th International Workshop, WABI 2008, Karlsruhe, Germany, September 15-19, 2008. Proceedings 8*. Springer, 2008, pp. 113–122.