Análise de Desempenho de uma Ferramenta para Visualização de Hashes em Dispositivos Móveis

Henrique Araújo de Carvalho, Jorge Miguel Ribeiro, Daniel Macêdo Batista, José Coelho de Pina

[1]Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo (USP)

henriquecarvalho@usp.br, jorgemiguelribeiro92@gmail.com

{batista, coelho}@ime.usp.br

Abstract. *A security application that enhances communication privacy must have a performance that ensures its usability without compromising the reservation of scarce resources, especially on mobile devices. Among such security applications, those such as hashify stand out, which manipulate or generate animations from cryptographic hashes. This paper presents the performance analysis of hashify, measuring the time taken to draw animations and evaluating memory and network consumption. Preliminary results obtained from an Android device showed that the images generated by the software are drawn, on average, in 22.6ms, a relatively low time showing that hashify can be integrated into mobile applications.*

Resumo. *Um aplicativo de segurança que aumente a privacidade de comunicações deve ter desempenho que garanta a sua usabilidade sem comprometer a reserva de recursos escassos, principalmente em dispositivos móveis. Dentre esses aplicativos, destacam-se aqueles, como o hashify, que manipulam ou geram animações a partir de hashes criptográficos. Este artigo apresenta a análise de desempenho do hashify, avaliando o tempo de geração de animações e consumo de memória e rede. Resultados preliminares obtidos com um dispositivo Android mostraram que a geração de imagens pelo software ocorre, em média, em 22,6ms, um tempo relativamente baixo e que mostra que ele pode vir a ser integrado em aplicativos móveis.*

1. Introdução

A comparação de *hashes* criptográficos é uma atividade cotidiana em segurança da informação e geralmente é atribuída a sistemas automatizados. Entretanto, muitas vezes, a garantia de segurança das comunicações pela Internet depende da comparação desses *hashes* por pessoas. Isso ocorre, por exemplo, na comparação de chaves criptográficas em aplicativos de mensagens e na comparação de certificados digitais de *websites*. Este trabalho realiza o exame de uma ferramenta [1] que facilita a comparação desses *hashes* por

[1]O código da ferramenta, chamada `hashify`, foi disponibilizado sob a licença MIT e está disponível em `https://gitlab.com/jorgemiguelribeiro92/hashify/`

pessoas, avaliando a viabilidade de sua utilização e de sua implementação em aplicativos Android.

O restante deste artigo está organizado da seguinte forma: A Seção 2 resume os trabalhos relacionados e apresenta a motivação para este artigo. A Seção 3 resume o funcionamento do `hashify`. A Seção 4 descreve a metodologia utilizada na análise de desempenho. A Seção 5 apresenta os resultados obtidos. O artigo é finalizado com a apresentação das conclusões e dos próximos passos na Seção 6.

2. Trabalhos Relacionados e Motivação

Ao longo das últimas duas décadas, vários estudos propuseram e examinaram ferramentas com a finalidade de facilitar a comparação de *hashes* criptográficos por pessoas, que são reconhecidamente ruins nessa tarefa [Perrig and Song 1999]. Enquanto nesses estudos os autores encontraram dificuldades na adoção de um sistema de geração de *hashes* visuais, a ferramenta `hashify`[2] busca superar essas dificuldades, avançando o estado da arte na comparação de *hashes* criptográficos por meio da geração de animações de 2 segundos de duração [Ribeiro et al. 2020].

Por exemplo, em *Hash Visualization: a New Technique to improve Real-World Security (1999)* [Perrig and Song 1999], os autores propõem a utilização de visualização de hashes como uma primeira etapa de verificação de senhas, feita pelos usuários, e como validação de chaves criptográficas. Os autores propõem a utilização de um algoritmo desenvolvido por Andrej Bauer, conhecido como Random Art, para a criação de *hashes* visuais. O algoritmo usa uma string de bits como a semente para um gerador de números aleatórios, que constrói uma expressão que descreve a função que irá gerar a imagem, mapeando cada pixel a um valor.

O `hashify` representa uma evolução desta ideia, já que propõe a utilização de imagens de objetos para uso na comparação de chaves criptográficas, e não imagens geradas de forma caótica. Supõe-se ser mais fácil comparar objetos e cores a comparar imagens genéricas, que podem envolver diferentes contrastes e posições de linhas.

Em *Developing and testing SCoP – a visual hash scheme (2014)* [Maina Olembo et al. 2014], os autores desenvolveram e testaram SCoP, uma proposta de *hash* visual que poderia substituir a comparação de sequências alfanuméricas. Os esquemas propostos nesse estudo, submetidos a experimentos com pessoas, foram mal sucedidos se consideradas outras limitações para a utilização de *hashes* visuais no mundo real, indicando que seria mais difícil ainda de se comparar *hashes* visuais em situações em que usuários não têm acesso à imagem correta e apenas lembram dela.

No caso do `hashify`, propõe-se a sua utilização em conjunto com um método de certificação alternativo, como um *smartphone*, de modo que não seria necessário lembrar exatamente do resultado visual do *hash*.

Em *Can Unicorns Help Users Compare Crypto Key Fingerprints? (2017)* [Tan et al. 2017] os autores desenvolvem um experimento realizado com 661 pessoas, que envolve a simulação de um ataque *Man in the Middle* (MitM). Foi solicitado às pessoas para compararem *hashes* que foram sutilmente modificados, à semelhança de como

[2]Uma demonstração de funcionamento do `hashify` pode ser vista em `https://youtu.be/raFkfRRXBM0`.

um atacante poderia agir para interceptar comunicações. Os autores concluem que as representações gráficas tiveram um sucesso inconclusivo, enquanto representações textuais foram mais bem sucedidas na comparação de hashes criptográficos. Isso porque representações gráficas de *hashes*, em geral, estariam mais suscetíveis a terem detalhes e aspectos essenciais ignorados na comparação, apesar de permitirem comparações mais rápidas e fáceis.

Ao contrário dos algoritmos usados por este estudo, o `hashify` é objetivamente mais fácil de se comparar porque envolve figuras, com cores evidentes, movimentos específicos e contendo um pequeno texto, o que também ajuda na orientação de pessoas sobre como se deve comparar *hashes* visuais.

Portanto, o `hashify` é uma alternativa bem sucedida às demais ferramentas que se propõem a facilitar a comparação de *hashes* criptográficos. Resta analisar o desempenho do `hashify` em dispositivos móveis, com a finalidade de verificar sua viabilidade como uma ferramenta de segurança nesses dispositivos. Essa análise é importante pelo fato dos dispositivos possuírem baterias como suas fontes principais de energia, o que justifica estudar o impacto que a ferramenta tem no consumo de recursos que possa levar a um consumo acelerado da bateria. Esse tipo de preocupação, relacionado à "complexidade energética" de algoritmos e componentes de software, tem sido discutida nos últimos anos na literatura especializada [Roy et al. 2013][Jain et al. 2005].

Este artigo apresenta os resultados preliminares alcançados na análise de desempenho do `hashify`. São avaliados tempo de geração das animações e consumo de memória e rede. Os resultados obtidos mostram que, a princípio, o `hashify` pode vir a ser integrado em aplicativos móveis. Por exemplo, em média, o tempo necessário para geração de animações foi de 22,6ms em um *smartphone* Samsung S20 FE, um dispositivo lançado no mercado em Outubro de 2020.

3. Funcionamento do hashify

O funcionamento do `hashify` é baseado em um algoritmo de geração de imagens com animações únicas derivadas a partir da composição de funções de *hash* tendo como argumento inicial o *hash* original. A Figura 1 mostra uma sequência com 5 quadros de uma animação gerada pelo `hashify`.

Na versão atual, o `hashify` pode ser utilizado embutido como uma extensão do navegador Firefox, o que permite que o mesmo seja utilizado em dispositivos móveis sem necessidade de alteração do seu código-fonte.

4. Metodologia

Um aplicativo de segurança que aumente a privacidade de comunicações deve ter desempenho que garanta a sua usabilidade sem comprometer a reserva de recursos escassos, principalmente em dispositivos móveis. Por este motivo, o presente estudo analisa, de forma estruturada, o desempenho do `hashify`, implementado como extensão no navegador Firefox, em execução em um *smartphone*.

A análise de desempenho foi realizada por meio da coleta de rastros. Rastros são arquivos que guardam informações sobre os processos executados no sistema operacional. Esses rastros contêm informações sobre eventos que ocorreram durante a execução de

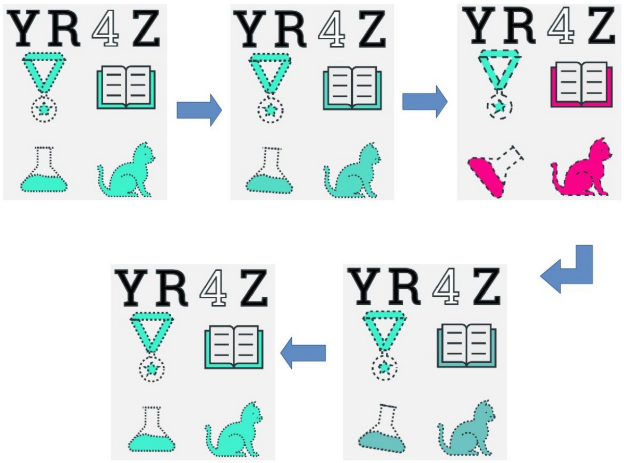

Figura 1. Quadros da animação gerada pelo `hashify` para o *hash* e0d31d7599daebee65a15a636736d65dae6963d2.

uma aplicação. Os eventos são associados a uma trilha específica, que funciona como uma linha do tempo crescente monotonicamente. Uma trilha corresponde a uma sequência independente de execução, como uma única *thread* em um processo [Google 2021b].

Os tipos de eventos analisados foram fatias e contadores. Fatias cobrem o período de tempo desde a chamada de uma função até seu término. Contadores são valores numéricos variáveis mostrados em cada instante de tempo.

A coleta dos rastros foi realizada com o `hashify`, instalado como uma extensão do Firefox, aberto em um *smartphone* e com uma única interação com o *smartphone*: um clique no botão de gerar animação. Todas as coletas de rastros foram realizadas em um Samsung modelo Galaxy S20 FE, exynos990, equipado com o Sistema Operacional Android 11 e com Firefox Browser Nightly 92.0a1.

Para a coleta dos rastros, foi usado o aplicativo `Rastreamento de sistema`, executado no próprio Android, que salva a atividade do dispositivo em um arquivo de rastreamento no formato perfetto-trace [Google 2020b]. Para a visualização e organização destes rastros, foi usada a ferramenta `Perfetto` [Google 2021a]. Além de permitir a visualização dos dados por meio de uma linha do tempo, o `Perfetto` exibe o tempo tomado pelos eventos de uma *thread* em uma tabela. Todos os dados quantitativos foram coletados dessa tabela. Usando essas ferramentas, foram coletados 25 rastros que permitiram uma análise estatística de informações sobre tempo de geração das animações, consumo de memória e consumo de rede.

Para que se possa comparar o desempenho do `hashify` a uma aplicação similar, também foram coletados 5 rastros durante a execução do software `RoboHash` [Davis 2011], uma aplicação com funcionamento similar ao do `hashify`.

5. Análise de Desempenho

Nesta seção, avalia-se o desempenho do `hashify` por meio dos rastros coletados. Esse desempenho foi medido através do tempo de geração de quadros, do consumo de memória e do consumo de rede. Ao final de cada uma das próximas subseções, o desempenho do `hashify` foi comparado com o desempenho do software `RoboHash`.

5.1. Tempo de Geração de Quadros

Um baixo tempo de geração de quadros é essencial para que uma aplicação de dispositivo móvel não tenha uma taxa de quadros instável ou para que o usuário não perceba uma alta latência da aplicação, o que é conhecido por instabilidade. Para que não haja instabilidade, a documentação do Android sugere que a geração de imagens e sua composição sejam feitas em um intervalo de 16ms, ou seja, os quadros precisam ser gerados e compostos a uma frequência de aproximadamente 60Hz. Caso a geração de quadros não aconteça a tempo, é possível que o desenho seja atrasado em alguns milissegundos. É deste atraso que vem a instabilidade experimentada por um usuário [Google 2020a].

Para saber avaliar o desempenho da aplicação neste quesito, antes de visualizar os dados do tempo de geração de quadros, é preciso entender o comportamento das *threads* responsáveis pela geração de quadros no Android.

A *UI Thread* (*thread* da interface do usuário) é a *thread* responsável pela geração e composição de imagens. É essa *thread* que recebe e avalia entrada, desenha a interface e organiza a execução de outros eventos de tela. Em geral, é a *thread* principal de um aplicativo Android. A *RenderThread* tem a função de rodar computações da GPU. Ela é responsável pela renderização de animações. A *thread SurfaceFlinger* é responsável por consumir as superfícies desenhadas por outros processos e compô-las em um único quadro a ser mostrado na tela. Finalmente, o *Choreographer* é o evento da *UI Thread* responsável por coordenar o tempo dessas *threads*.

A Figura 2 permite visualizar o trabalho das *threads* descritas acima. O evento *Choreographer* (linha em marrom na figura) é chamado na *UI Thread* (identificada como "AndroidUI" à esquerda na figura) e, então, o trabalho da CPU se inicia na construção dos quadros até quando a *RenderThread* é chamada para renderizar as imagens. Como pode ser visto na figura, ambas as *threads* são chamadas duas vezes seguidas, de modo que consideramos o final da segunda vez para o marco do término da geração dos quadros. Nesta amostra, tudo tem duração aproximada de 19,2ms.

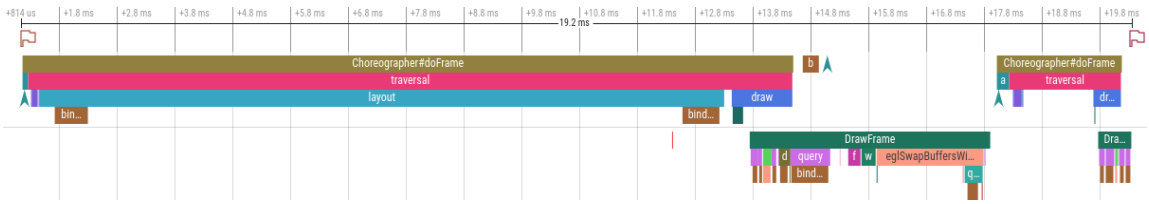

Figura 2. Linha do tempo de execução (Saída do software `Perfetto`)

Apesar dessa amostra ter uma composição que demora aproximadamente 3,2ms a mais que o limite sugerido de 16ms, entendemos que isso não seria suficiente para que o usuário sinta a instabilidade. Isso porque o atraso ocorre durante um único quadro e a frequência de geração de quadros, a um tempo de aproximadamente 20ms, é de 50Hz e continua alta e, possivelmente, indetectável por pessoas. Assim, entendemos que o `hashify` tem um desempenho de tempo de geração de quadros satisfatório se ele se mantiver abaixo de 20ms. Pelos dados coletados (Figura 3), foi possível verificar que isso ocorre em pelo menos 50% das vezes, uma vez que a mediana da duração da geração de quadros foi 19,2ms. A média da duração foi 22,6ms para os dados obtidos, com um

desvio padrão de 6,4ms. Ou seja, em geral o `hashify` performa muito bem em tempo de geração de quadros e não deve atrapalhar a experiência do usuário com instabilidade.

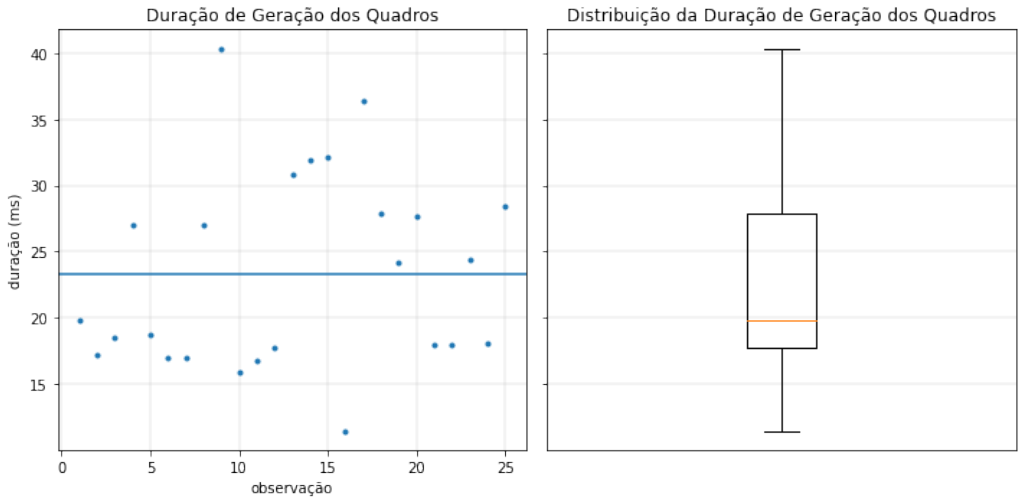

Figura 3. Observações do tempo de execução

A título de comparação, o `RoboHash` levou em média 28ms para gerar uma imagem, o que significa que o `hashify` possui desempenho médio, em termos de geração de quadros, cerca de 19,3% melhor que o deste software.

5.2. Consumo de Memória

Alguns contadores presentes no rastro são capazes de dar uma ideia sobre o consumo de memória pelo `hashify`. A Figura 4 permite visualizar os dados de memória que são mostrados pelo rastro. Esse corte do rastro foi feito exatamente durante a geração dos quadros pelo `hashify`.

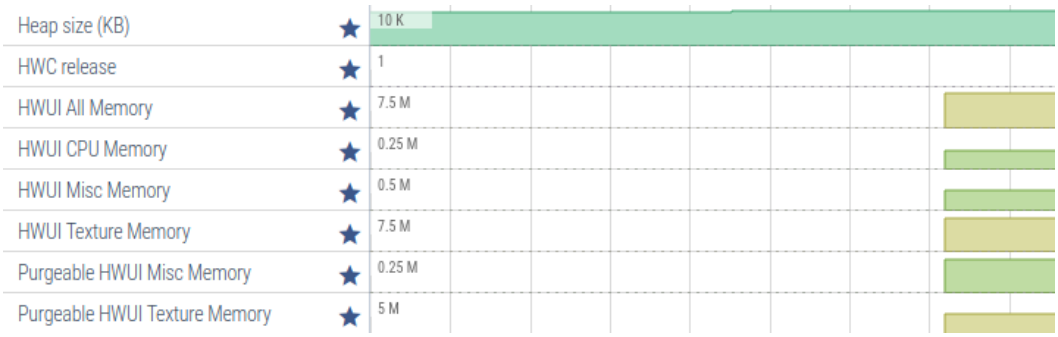

Figura 4. Consumo de memória (Saída do software `Perfetto`)

Pode-se notar que o rastro dispõe duas memórias diferentes para a execução da aplicação, o *heap* e a memória HWUI. Para esta última, além do contador que conta a memória total HWUI, existem outros contadores que mostram a memória HWUI segmentada.

Enquanto a memória do *heap* mede a memória alocada na RAM durante a execução da aplicação, memória total HWUI é a soma de toda a memória alocada para caches (cache de textura, de camadas, de gradiente, etc) da biblioteca Android HWUI,

que permite que componentes da interface do usuário sejam acelerados pela GPU. É por este motivo que no rastro da figura o contador dessa memória sobe imediatamente após o evento *DrawFrame*, da *RenderThread*, terminar seu trabalho. Portanto, para esta análise interessa a memória do *heap* e a memória total HWUI.

Sobre a memória do *heap*, interessa para a análise a variação do tamanho dessa memória. Considera-se variação de memória do *heap* como a diferença entre o valor anterior e o valor posterior à geração de quadros. Esses dados foram dispostos na Figura 5. Neste caso, a mediana das observações é bem próxima da média, o que indica que a variação da memória do *heap* tem uma distribuição normal. Em média existe um consumo de aproximadamente 82,7KB de memória do *heap*, com um desvio padrão de 23,0KB, e espera-se que, em pelo menos 50% das vezes, seja consumido 80,0KB ou menos.

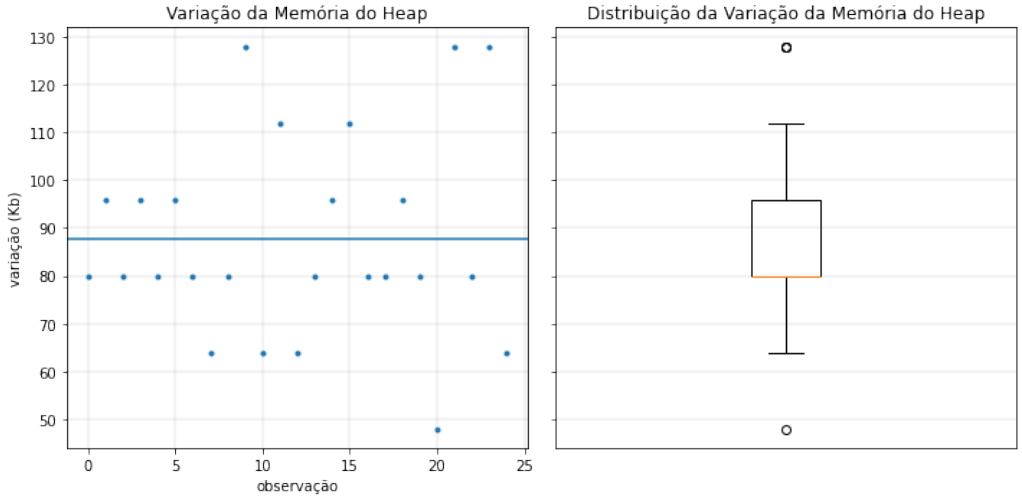

Figura 5. Observações do consumo de memória do *heap*

Sobre a memória total HWUI, interessa para a análise o tamanho dessa memória após o evento *DrawFrame* terminar. Esse valor é constante para todas as execuções, 7,371860MB. A título de comparação, após a execução do software `RoboHash`, a memória total HWUI tem valor de 4,365162MB. Embora o consumo do `hashify` seja maior, vale observar que a ferramenta gera uma animação e não uma imagem estática.

5.3. Consumo de Rede

NetworkPolicy é o processo que chama um conjunto de eventos de rede. Entretanto, não foi identificado qualquer consumo de rede pelo `hashify`, de modo que é possível criar animações com a extensão mesmo sem acesso à Internet.

Este resultado era esperado, uma vez que a extensão foi carregada localmente e executada no próprio *smartphone*. De toda forma, a análise se faz necessária, diante da possibilidade das bibliotecas usadas no desenvolvimento da aplicação poderem acessar recursos de rede. Ainda, uma vez que o `RoboHash` está acessível unicamente por seu *website*, não foi possível realizar a comparação deste quesito entre as duas aplicações.

6. Conclusão e Próximos Passos

Este artigo apresentou a análise de desempenho do `hashify`, uma aplicação de geração de animações a partir de *hashes*, com a finalidade de avaliar a viabilidade de sua

implementação em aplicativos de segurança em dispositivos móveis.

Após análise preliminar de rastros obtidos durante a execução do software, conclui-se que o software possui bom desempenho durante a geração das animações e que é, portanto, uma alternativa bem sucedida às demais ferramentas que propuseram a facilitar a comparação de *hashes* criptográficos no passado. Dessa forma, entende-se que o `hashify` pode vir a ser integrado em aplicações de dispositivos móveis sem comprometer o desempenho dessas aplicações. O trabalho encontra-se em andamento. Como próximos passos, pretende-se analisar o consumo de bateria do `hashify`, repetir os experimentos em pelo menos outro modelo de *smartphone* e realizar experimentos com pessoas, a fim de testar a eficácia do `hashify`.

Agradecimentos

Esta pesquisa é parte do INCT da Internet do Futuro para Cidades Inteligentes, financiado por CNPq (proc. 465446/2014-0), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001 e FAPESP (procs. 14/50937-1 e 15/24485-9). Também é parte dos projetos FAPESP proc. 18/22979-2, 18/23098-0 e 21/04266-1.

Referências

Davis, C. (2011). RoboHash. `https://robohash.org/`. Acessado em 23/07/2021.

Google (2020a). Renderização lenta. `https://developer.android.com/topic/performance/vitals/render`. Acessado em 22/07/2021.

Google (2020b). Visão geral do rastreamento do sistema. `https://developer.android.com/topic/performance/tracing`. Acessado em 12/07/2021.

Google (2021a). Perfetto - System profiling, app tracing and trace analysis. `https://perfetto.dev/`. Acessado em 12/07/2021.

Google (2021b). Track events (Tracing SDK). `https://perfetto.dev/docs/instrumentation/track-events`. Acessado em 22/07/2021.

Jain, R., Molnar, D., and Ramzan, Z. (2005). Towards a Model of Energy Complexity for Algorithms [Mobile Wireless Applications]. In *IEEE Wireless Communications and Networking Conference, 2005*, volume 3, pages 1884–1890.

Maina Olembo, M., Kilian, T., Stockhardt, S., Hülsing, A., and Volkamer, M. (2014). Developing and Testing SCoP–a Visual Hash Scheme. *Information Management & Computer Security*, 22(4):382–392.

Perrig, A. and Song, D. (1999). Hash Visualization: a New Technique to Improve Real-World Security. In *CryTEC '99*.

Ribeiro, J. M., Batista, D. M., and de Pina, J. C. (2020). hashify: Uma Ferramenta para Visualização de Hashes com Animações. In *Anais do Salão de Ferramentas do SBSeg*.

Roy, S., Rudra, A., and Verma, A. (2013). An Energy Complexity Model for Algorithms. In *4th Conference on Innovations in Theoretical Computer Science*, page 283–304.

Tan, J., Bauer, L., Bonneau, J., Cranor, L. F., Thomas, J., and Ur, B. (2017). Can Unicorns Help Users Compare Crypto Key Fingerprints? In *ACM CHI'17*, pages 3787–3798.