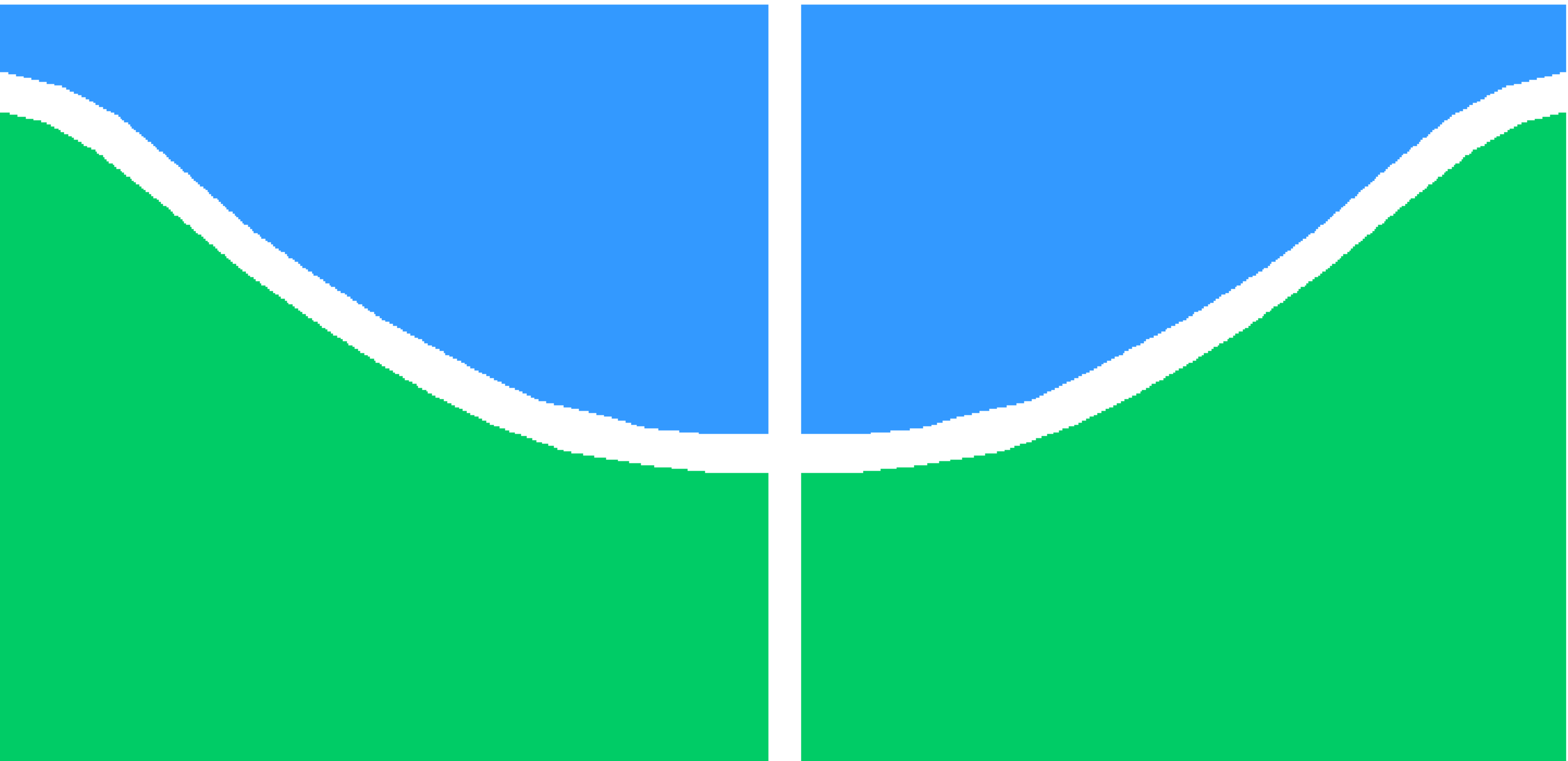



**Universidade de Brasília – UnB**

**Faculdade UnB Gama – FGA**

**Engenharia de software**


# Kiskadee: Sistema de monitoramento e análise contínua de código fonte


**Autor: David Carlos de Araújo Silva**

**Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles**



**Brasília, DF**

**2017**

David Carlos de Araújo Silva

# Kiskadee: Sistema de monitoramento e análise contínua de código fonte

Monografia submetida ao curso de graduação em Engenharia de software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Coorientador: Athos Coimbra Ribeiro

Brasília, DF

2017

David Carlos de Araújo Silva

Kiskadee: Sistema de monitoramento e análise contínua de código fonte/ David Carlos de Araújo Silva. – Brasília, DF, 2017-

63 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2017.

1. analise. 2. monitoramento. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Kiskadee: Sistema de monitoramento e análise contínua de código fonte

CDU 02:141:005.6

David Carlos de Araújo Silva

# Kiskadee: Sistema de monitoramento e análise contínua de código fonte

Monografia submetida ao curso de graduação em Engenharia de software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de software.

Trabalho aprovado. Brasília, DF, 23 de março de 2017:

---

**Prof. Dr. Paulo Roberto Miranda Meirelles**
Orientador

---

**Dr. Antonio Terceiro**
Convidado 1

---

**Me. Phyllipe Lima**
Convidado 2

Brasília, DF
2017

# Resumo

Desenvolver software é uma atividade criativa e por ser realizada por seres humanos está sujeita a defeitos. Dito isso, são diversos os defeitos que podem ocorrer em um projeto de software, mas os que mais nos interessam são aqueles relacionados ao produto do projeto, ou seja, ao código fonte. Defeitos no código fonte podem ser denominados como $fraquezas$ que, dependendo de suas características, podem levar a vulnerabilidades de segurança. Em projetos de médio e grande porte encontrar tais fraquezas inspecionando manualmente todos os módulos do projeto é inviável, além disso, projetos ativos lançam com certa frequência novas versões que podem vir com fraquezas não identificadas pelo time de desenvolvimento. Este trabalho foi construído no contexto de análise estática e teve por objetivo desenvolver um sistema de monitoramento e análise contínua de código fonte, permitindo aos desenvolvedores e pesquisadores coletarem dados a respeito da qualidade de projetos diversos ao longo do tempo. Como resultado, uma ferramenta denominada kiskadee foi criada, capaz de monitorar diferentes repositórios de código fonte, realizando análises estáticas a cada nova versão lançada pelos projetos monitorados.

**Palavras-chaves**: Análise estática, código fonte, fraquezas

# Abstract

Developing software is a creative activity. Since it is carried out by humans, it is subject to defects. With this in mind there are a few of types of defects that can occur in a software project, but the ones that interest us most, are those related with the product of the project, that is, the source code. Source code defects can be named as *weaknesses*, which depending on their characteristics, can lead to security vulnerabilities. In medium and large projects, finding such weaknesses by manually inspecting all project modules is not pratical, in addition to the fact that active projects launch with some frequency new versions that may come with new weaknesses not identified by the development team. This work is built within the context of static analysis, and aims to develop a system of continuous monitoring and analysis of source code, allowing developers and researchers to collect data about quality of different projects over time. As result, a tool called kiskadee was created, able to monitors different source code repositories, doing static analysis in each released version by the monitored projects.

**Key-words**: Static analysis, source code, software flaws

# Lista de ilustrações

Figura 1 – Visão genérica do processo de análise estática . . . . . . . . . . . . . . 23
Figura 2 – Exemplo de árvore sintática . . . . . . . . . . . . . . . . . . . . . . . . 24
Figura 3 – Arquitetura do kiskadee . . . . . . . . . . . . . . . . . . . . . . . . . . 29
Figura 4 – Comunicação com o Anitya . . . . . . . . . . . . . . . . . . . . . . . . 40
Figura 5 – Modelo de domínio do kiskadee . . . . . . . . . . . . . . . . . . . . . . 42
Figura 6 – Fluxo de monitoramento de uma distribuição . . . . . . . . . . . . . . 43
Figura 7 – Fluxo de Monitoramento de um upstream . . . . . . . . . . . . . . . . 44
Figura 8 – Lista de pacotes srepositórios pelo kiskadee . . . . . . . . . . . . . . . 45
Figura 9 – Versão analisada do pacote 0xffff . . . . . . . . . . . . . . . . . . . . . 45
Figura 10 – análise do pacote 0xffff, feita pelo Cppcheck . . . . . . . . . . . . . . . 46
Figura 11 – análise do pacote 0xffff, feita pelo Flawfinder . . . . . . . . . . . . . . . 46
Figura 12 – Nova versão do projeto Xe . . . . . . . . . . . . . . . . . . . . . . . . 47
Figura 13 – Evento de mudança de versão . . . . . . . . . . . . . . . . . . . . . . . 47
Figura 14 – Trecho da mensagem publicada no fedmsg . . . . . . . . . . . . . . . . 47
Figura 15 – Pacote e Versão do projeto Xe . . . . . . . . . . . . . . . . . . . . . . 48
Figura 16 – Análise feita no projeto Xe, pelo Cppcheck . . . . . . . . . . . . . . . . 48
Figura 17 – Análise feita no projeto Xe, pelo Flawfinder . . . . . . . . . . . . . . . 49
Figura 18 – Estrutura do Debile . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

# Lista de tabelas

Tabela 1 – Taxa de falsos negativos das ferramentas (CHAHAR et al., 2012) . . . 23
Tabela 2 – Taxa de falsos positivos das ferramentas (CHAHAR et al., 2012) . . . . 23

# Lista de abreviaturas e siglas

GSoC *Google Summer of Code*: Programa de verão patrocinado pelo Google, para fomentar o desenvolvimento de projetos de código aberto, por parte de alunos de universidades ao redor do mundo.

CWE *Common Weakness Enumeration*: Dicionário de fraquezas comuns de ocorrerem no código fonte.

NIST *National Institute of Standards and Technology*

# Sumário


**1 INTRODUÇÃO . . . 17**

**2 ANÁLISE ESTÁTICA E FRAQUEZAS DE SOFTWARE . . . 21**
**2.1 Analisadores estáticos . . . 23**

**3 PROPOSTA DE FERRAMENTA DE MONITORAMENTO . . . 27**
**3.1 Trabalhos relacionados . . . 27**
**3.2 Kiskadee . . . 28**
3.2.1 Repositórios . . . 30
3.2.2 Fetchers . . . 30
3.2.3 Monitor . . . 32
3.2.4 Executor . . . 32
3.2.5 Analisadores Estáticos . . . 33

**4 IMPLEMENTAÇÃO . . . 35**
**4.1 Fetchers . . . 35**
4.1.1 Debian . . . 37
4.1.2 Anitya . . . 38
4.1.3 Juliet . . . 40
**4.2 Analisadores Estáticos . . . 40**
**4.3 Modelo de domínio . . . 41**

**5 EXEMPLO DE USO . . . 43**

**6 CONSIDERAÇÕES FINAIS . . . 51**
**6.1 Trabalhos Futuros . . . 52**

**APÊNDICES 53**

**APÊNDICE A – REVISÃO DA ARQUITETURA DO DEBILE . . . 55**
A.0.1 Arquitetura . . . 55
A.0.1.1 Runners . . . 56

**APÊNDICE B – CATÁLOGO DE FRAQUEZAS DE SOFTWARE . . . 57**

**APÊNDICE C – ARQUIVO DE CONFIGURAÇÃO DO KISKADEE 59**

APÊNDICE D – TRECHO DO ARQUIVO SOURCES.GZ . . . . . . 61

REFERÊNCIAS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

# 1 Introdução

A identificação automatizada de fraquezas de software que levam à vulnerabilidades de segurança tem sido o objeto de estudo de grandes instituições como o $NIST$[1] (National Institute of Standards and Technology). O esforço em coletar, padronizar e priorizar fraquezas com potencial para se tornarem vulnerabilidades exploráveis motivou, por exemplo, projetos como $SARD$ ( Software Assurance Reference Dataset)[2], que buscou construir uma base de dados de fraquezas a partir da coleta de diferentes ferramentas de análise estática. O projeto SARD teve outro objetivo que não só a coleta de grandes casos de teste de fraquezas, mas também encorajar a adoção de ferramentas de análise estática por parte dos desenvolvedores.

A decisão entre utilizar ou não uma ferramenta de análise estática para identificar possíveis fraquezas em um código fonte pode ser influenciada pelo tamanho do projeto. A coleta manual por fraquezas não escala para grandes bases de código, ou seja, conforme o projeto cresce, realizar tal coleta sem o auxílio de uma ferramenta de análise estática se torna uma prática ineficiente. Ao optar pelo uso de ferramentas de análise estática, surge a necessidade de priorizar quais fraquezas identificadas deverão ser solucionadas, uma vez que poucas organizações possuem os recursos necessários para inspecionar cada uma dessas fraquezas. Cabe aos desenvolvedores direcionar esforços para as fraquezas mais críticas (KHAZAEI et al., 2016). O projeto $CWSS$[3] (Common Weakness Scoring System), propõe um sistema de priorização para fraquezas de software, permitindo incluir informações relacionadas ao ambiente em que dada fraqueza foi identificada.

Ferramentas diferentes analisam o código fonte de maneiras diferentes, isso implica que certas fraquezas podem passar desapercebidas por uma ferramenta e não por outra. Combinar diferentes ferramentas pode dar ao desenvolvedor uma visão mais completa da qualidade do código avaliado, mesmo sabendo que por ser uma análise estática uma série de aproximações em relação ao comportamento do código fonte serão feitas. Isso ocorre pelo fato da ferramenta não saber o comportamento do código ao ser executado. Um dos obstáculos em utilizar diferentes ferramentas para identificar um maior número de fraquezas em um código fonte, reside no fato de que teremos diferentes formatos de relatórios, o que dificulta a comparação entre as análises. Por exemplo, uma mesma fraqueza que foi identificada por duas ferramentas diferentes mas que em cada resultado, tal fraqueza é apresentada de maneira distinta.

Atualmente, é comum projetos de software serem desenvolvidos de maneira aberta

[1] https://www.nist.gov/
[2] https://samate.nist.gov/SRD/
[3] https://cwe.mitre.org/cwss/cwss_v1.0.1.html

e descentralizada. Dependendo da licença utilizada para guiar aspectos de uso, distribuição, alteração e comercialização do projeto, o código fonte poderá estar disponível em repositórios de acesso público. No contexto deste trabalho utilizaremos o termo em inglês *upstream*, quando fizermos referência ao responsável pela manutenção e lançamento de novas versões de determinado código fonte. Um repositório aberto em sistemas como o GitHub[4] pode ser considerado um *upstream*. Além do *upstream* iremos lidar também com distribuições Linux, como o Debian[5] e o Fedora[6]. Uma das grandes vantagens em monitorar essas distribuições é que todos os pacotes disponibilizados oficialmente ao usuário final terão os seus respectivos códigos fonte também disponíveis, podendo ser analisados conforme nossas necessidades. Ao final desse trabalho nosso objetivo é desenvolver um sistema capaz de monitorar e analisar o maior número de projetos possível.

Para que o monitoramento contínuo seja possível, entendemos que o acesso ao código fonte dos projetos monitorados tem de ser automatizado, e a complexidade em implementar o monitoramento de qualquer fonte de código (*upstream* ou distribuição Linux) seja definida apenas pelas particularidades da fonte em si. Entendemos também que a arquitetura da ferramenta deve suportar que cada desenvolvedor defina a forma como esse monitoramento deve ser feito. Neste contexto, partimos da seguinte questão-problema a ser tratada no presente trabalho:

*Como definir um sistema extensível para o monitoramento e análise estática contínua de código fonte?*

Nossa questão-problema é motivada pelo fato de que não existe um sistema de código aberto que permita monitorar diferentes projetos de software com o objetivo de automatizar a análise estática do código fonte. Partindo dessa constatação um sistema denominado kiskadee foi construído por nós. Esse sistema é responsável por monitorar continuamente projetos de software a fim de realizar análises estáticas no código fonte. Isso irá permitir que desenvolvedores tenham acesso a *warnings* diversos, sem a necessidade de se preocupar em configurar e executar ferramentas, a cada nova versão lançada pelo do projeto monitorado.

Este trabalho está participando de um programa de desenvolvimento financiado pelo Google, chamado Google Summer Of Code (GSoC). Nesse programa, centenas de alunos de universidades espalhadas pelo mundo, são mentorados por organizações financiadas pelo Google, com o objetivo de fomentar o desenvolvimento de soluções de código aberto e soluções que sejam software livres. O kiskadee está sendo mentorado pelo projeto Fedora, em específico pelo desenvolvedor Athos Ribeiro. No endereço <https://goo.gl/E7z8vP> é possível ler a proposta que submetemos para o programa.

[4] https://github.com/
[5] https://www.debian.org/
[6] https://getfedora.org/

Este trabalho de conclusão de curso está organizado em 5 capítulos. No Capítulo 2 faremos uma breve revisão bibliográfica a respeito do que é análise estática de código fonte, seu papel no ciclo de desenvolvimento e o que existe do ponto de vista de ferramentas para coleta de fraquezas de software. No Capítulo 3 abordaremos a proposta arquitetural do kiskadee. No Capítulo 4, demonstraremos de que maneira a ferramenta foi implementada. No Capítulo 5, apresentaremos a validação de que a arquitetura proposta pôde ser utilizada para o monitoramento contínuo de código fonte. No Capítulo 6, apresentaremos a conclusão e trabalhos futuros envolvendo a ferramenta.

# 2 Análise estática e fraquezas de software

Segundo a IEEE, *análise estática* é o processo de avaliar um sistema ou componente baseando-se em sua forma, estrutura, conteúdo ou documentação. Esse conceito contrasta com a *análise dinâmica*, que avalia sistemas e componentes ao serem executados. Quando falamos na forma, estrutura e conteúdo de um sistema podemos afirmar que estamos nos referindo ao seu código fonte, mas sabemos que não existem apenas medições estáticas ou dinâmicas. Segundo FENTON et al., medições de software podem ser categorizadas de três formas:

- **Medição de projeto**: Medição relacionada a atividades do projeto e marcos.
- **Medição de processo**: Medição relacionada ao ciclo de vida do projeto.
- **Medição de produto**: Medição relacionada ao código fonte do projeto.

Para realizar uma medição de produto efetiva e automatizada, diversas ferramentas foram construídas para atender a diferentes necessidades relacionadas a qualidade do produto a ser avaliado. Essas ferramentas permitem ao desenvolvedor coletar informações a respeito de diferentes características do seu código fonte. Ao observar tais informações é possível atuar em pontos específicos do código, realizar melhorias e consertar possíveis problemas. Segundo levantamento feito por EMANUELSSON, ferramentas de análise estática podem ser utilizadas nos seguintes contextos:

- *Debug* de código fonte
- Geração automática de casos de teste
- Análise de impacto
- Detecção de intrusão
- Métricas de código fonte
- Captura de erros de codificação
- Captura de fraquezas de segurança
  - Divisão por zero
  - Uso improprio de recursos (alocação de memória, arquivos)
  - Operações ilegais

– Código incompleto

A medição de produto auxilia na captura de erros comuns de codificação como estouro de pilha, variáveis declaradas e não utilizadas, vazamento de memória e condições de leitura e escrita. Todos essas fraquezas podem levar a uma ou mais vulnerabilidades de segurança. Segundo CHAHAR et al., uma análise de código fonte efetiva possibilita observar o comportamento de um programa, com o objetivo de encontrar possíveis vulnerabilidades. Uma vulnerabilidade é uma propriedade de requisitos de sistemas de segurança, projeto, implementação, ou operação que poderia ser acidentalmente disparada ou intencionalmente explorada resultando em uma falha de segurança (OKUN et al., 2008). Ainda segundo OKUN et al., uma vulnerabilidade é o resultado de uma ou mais fraquezas nos requisitos, projeto, implementação ou operação. Um *warning* pode ser um problema normalmente relacionado a uma fraqueza identificada por uma ferramenta, presente no relatório gerado por esta ao ser executada no código fonte. Uma fraqueza pode levar a uma vulnerabilidade de segurança e no apêndice B demonstramos um exemplo de fraqueza que não leva a uma vulnerabilidade explorável.

A indecibilidade da análise estática de código fonte, faz com que ferramentas utilizadas para realizar a coleta de fraquezas realizem uma série de aproximações a respeito do real comportamento do código avaliado (LANDI, 1992). Isso quase sempre leva a falsos alarmes, por exemplo, *warnings* que não são correlacionados a nenhuma execução real (FEHNKER et al., 2010). Esse é um típico caso de *falso positivo*, que pode ser descrito como um *warning* reportado pela ferramenta, mas que não é uma fraqueza real no código. Segundo HECKMAN et al., o valor de falsos positivos reportados por ferramentas de análise estática pode variar entre 35-91%. Uma tentativa, por parte de algumas ferramentas para contornar falsos positivos, é adicionar níveis de severidade para cada *warning* reportado. Isso permite que o responsável por avaliar o relatório da ferramenta selecione apenas *warnings* com um alto grau de severidade.

O fato de que cada ferramenta avalia o código fonte de uma maneira particular, e ainda considerando o caráter indecidível da análise estática, faz com que certas fraquezas não sejam identificadas por determinada ferramenta, levando a não inclusão do *warning* no relatório da análise. Chama-se tal fato de *falso negativo*, e ocorre quando certas fraquezas no código não são identificadas por uma ferramenta criada para tal fim. Um dos objetivos do sistema de monitoramento e análise que propomos é reduzir o número de falsos negativos presentes em uma análise ao combinar diferentes ferramentas avaliando um mesmo código fonte (BLACK, 2009).

No estudo feito por CHAHAR et al., um comparativo entre a efetividade (taxa de falsos positivos e falsos negativos) de algumas ferramentas foi realizado. As ferramentas

avaliadas foram Cppcheck[1], Flawfinder[2] e Splint[3]. Os resultados encontrados podem ser observados nas Tabelas 1 e 2.

Tabela 1 – Taxa de falsos negativos das ferramentas (CHAHAR et al., 2012)

| | Flawfinder | Splint | Cppcheck |
|---|---|---|---|
| Número de casos de teste | 20 | 20 | 20 |
| fraquezas encontrados | 7 | 16 | 9 |
| fraquezas perdidas | 13 | 4 | 11 |
| % de falsos negativos | 65% | 20% | 55% |

Tabela 2 – Taxa de falsos positivos das ferramentas (CHAHAR et al., 2012)

| | Flawfinder | Splint | Cppcheck |
|---|---|---|---|
| Número de casos de teste | 21 | 21 | 21 |
| número de warnings | 8 | 4 | 2 |
| % de falsos positivos | 38.09% | 19.04% | 9.52% |

Os dados obtidos nesse comparativo demonstram o quanto a tarefa de analisar o código fonte de maneira estática está sujeita à um certo grau de incerteza. Observando a taxa de falsos positivos e negativos das ferramentas é possível identificar a dificuldade encontrada por muitos desenvolvedores em incluir tais ferramentas no processo de acompanhamento da qualidade do software. Um grande número de falsos positivos podem levar desenvolvedores e gestores a rejeitar o uso de análises estáticas automatizadas como parte do processo de desenvolvimento, devido ao retrabalho na inspeção desses falsos positivos (HECKMAN et al., 2011).

## 2.1 Analisadores estáticos

Podemos observar um analisador estático como um caixa fechada, que a partir de uma dada entrada $X$, nos gera uma saída $Y$. Essa entrada $X$ é o código fonte a ser avaliado e a saída $Y$ o relatório com *warnings* indicando possíveis problemas encontrados. A Figura 1 demonstra de maneira genérica o funcionamento de uma ferramenta de análise estática (CHAHAR et al., 2012).

A etapa de construção do modelo apresentado na Figura 1 é responsável por transformar o código fonte em uma série de símbolos, descartando características irrelevantes do código como espaços em branco e comentários. Para ferramentas como o Cppcheck, este passo é chamado de *pré-processamento*. O conjunto de símbolos são então organizados em uma árvore sintática por meio de um *parser*. O parser identifica cada *token* a partir de uma gramática livre de contexto ou uma gramática *context-sensitive*. É a gramática que

[1] http://cppcheck.sourceforge.net/
[2] https://www.dwheeler.com/flawfinder/
[3] http://www.splint.org/

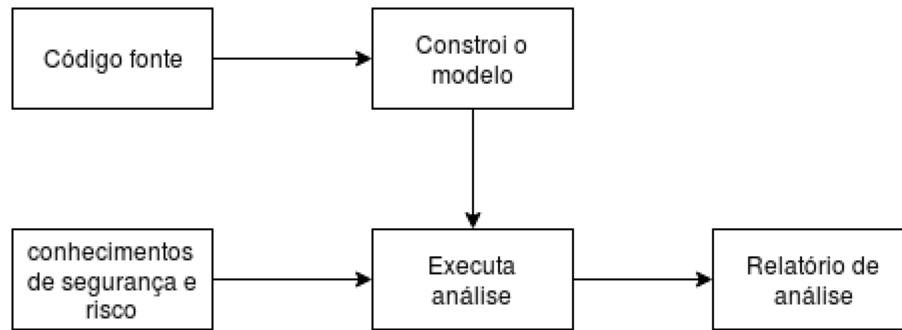



Figura 1 – Visão genérica do processo de análise estática

dá significado para cada símbolo que vem do pré-processamento (CHAHAR et al., 2012). É analisando a árvore sintática que ferramentas como o Cppcheck identificam possíveis fraquezas no código fonte. Um exemplo de árvore sintática é apresentado na Figura 2.

$$x = a + b;$$

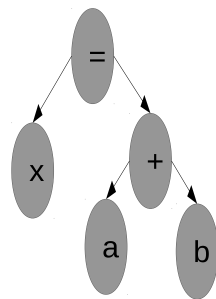



Figura 2 – Exemplo de árvore sintática

É a partir da gramática que a ferramenta toma conhecimento de como estruturar a árvore sintática de determinado conjunto de símbolos que estão sendo processados. Antes de realizar qualquer checagem na árvore, existe um passo anterior que é avaliar o valor dos símbolos identificados como variáveis, e então armazenar esse valor junto a variável dentro da árvore sintática. Isso permite, por exemplo, que as ferramentas identifiquem fraquezas como variáveis não inicializadas, divisões por zero e assim por diante. Vale ressaltar que utilizar uma árvore sintática é apenas uma das possíveis abordagens para a construção do modelo, apresentado na Figura 1.

Após executar a análise propriamente dita, a ferramenta encerra seu funcionamento gerando um relatório com os *warnings* encontrados. Esse relatório pode ser gerado de diversas formas, desde utilizando a saída padrão até utilizando um arquivo com formato XML ou CSV. Cada ferramenta irá gerar um relatório de acordo com o proposito para a qual ela foi construída, e essa não conformidade entre relatórios dificulta o uso de diferentes ferramentas avaliando um mesmo código fonte. Uma opção dada, por exemplo, pela

ferramenta Cppcheck, é incluir, em cada *warning* gerado, um identificador, relacionado a um banco de fraquezas catálogadas por diversas organizações da industria e da academia, sendo utilizado como referência na área de segurança de software. Esse catálogo chama-se *CWE* (Common Weakness Enumeration)[4]. É importante frisar que nem todas as ferramentas de análise estática utilizam identificadores CWE em seus relatórios. No apêndice B fizemos uma breve revisão sobre esse projeto.

[4] http://cwe.mitre.org/

# 3 Proposta de ferramenta de monitoramento

Neste capitulo apresentaremos a proposta de arquitetura do kiskadee. Como passo inicial devemos relembrar nossa questão-problema:

*Como definir um sistema extensível para o monitoramento e análise estática contínua de código fonte?*

Este trabalho acadêmico pode ser classificado, do ponto de vista de sua natureza como aplicado, uma vez que a partir dos conhecimentos bibliográficos iremos propor a construção de um sistema que permita a coleta contínua de fraquezas de software, a partir da análise de projetos hospedados em repositórios públicos.

## 3.1 Trabalhos relacionados

RENATUS et al. propõe uma ferramenta de pontuação para fraquezas de software utilizando tanto o projeto CWE quanto o projeto *Common Vulnerability Scoring System*[1] (CVSS), permitindo agregar mais valor ao relatório gerado por ferramentas de análise estática, uma vez que muitas variáveis de ambiente que influenciam na coleta de fraquezas não podem ser automaticamente coletadas.

*Static Analysis Tool Exposition* (SATE) foi um projeto liderado pelo NIST, que teve por objetivo criar uma base de dados com fraquezas de software coletadas por diferentes ferramentas de análise estática. O projeto teve foco em ferramentas que, avaliando o código fonte de um conjunto de projetos de código aberto pudessem detectar e reportar fraquezas de software. Vale ressaltar que a submissão de cada ferramenta foi feita utilizando o formato XML, seguindo um modelo definido pelo próprio projeto.

*ClabureDB*[2] é um banco de dados que pode ser utilizado como uma ferramenta para customizar e avaliar ferramentas de análise estática. O banco de dados contem relatórios de erros produzidos por diferentes ferramentas, além de análises manuais feitas por pesquisadores envolvidos no projeto. Esses relatórios são classificados tanto como erros reais ou falsos positivos. Atualmente esse banco de dados contem mais de 800 relatórios de *bug* detectados no kernel linux 2.6.28.

*Debile*[3] foi um projeto criado para atender uma demanda específica da distribuição GNU/linux Debian, sendo descrito como um sistema de *build*. É pequeno, minimo e implementado utilizando Python. Isso permite que qualquer um use o Debile como uma

[1] https://www.first.org/cvss
[2] http://decibel.fi.muni.cz:3000
[3] https://github.com/opencollab/debile

plataforma de build de pacotes com formato .deb, ou executando ferramentas customizadas tanto no código fonte quanto nos pacotes propriamente ditos. O Debile foi de grande importância para este trabalho e em um primeiro momento foi utilizado para realizar algumas análises estáticas no código fonte do kernel Linux. Devido ao alto acoplamento do Debile à infraestrutura do Debian, não foi possível evoluí-lo para um contexto mais genérico de monitoramento e isso foi o que de fato motivou a criação do kiskadee. Ainda assim o Debile foi fonte de inspiração para a implementação que será descrita nesse capítulo, e no apêndice A fizemos um breve resumo sobre sua arquitetura.

Firehose é um pacote Python que tem por objetivo gerenciar o resultado de ferramentas de análise estática. Ele provê *parsers* para a saída do GCC[4], Clang-Analyzer[5], Cppcheck e Findbugs[6]. Esses *parsers* convertem os resultados em um modelo comum de objetos Python, provendo um arquivo em formato XML como resultado. O projeto *mock-with-analysis*[7] injeta diversos analisadores estáticos durante a reconstrução de um pacote rpm (formato de pacote para distribuições Linux como o Fedora) e converte o resultado das análises para um formato Firehose. O kiskadee utiliza os *parsers* disponibilizados pelo Firehose para padronizar o formato com que os analisadores geram os relatórios de análise. Coala[8] é um projeto que possui uma proposta similar ao Firehose, mas padronizar o relatório de diferentes ferramentas de análise estática é apenas uma das funcionalidades oferecidas. O projeto permite correção automática de erros, integração com editores de texto e diversas outras funcionalidades que podem ser exploradas na documentação do projeto.

Levando em conta os trabalhos relacionados listados previamente e a nossa necessidade em realizar um monitoramento contínuo de fraquezas de software, não foi possível encontrar um sistema que permita monitorar continuamente códigos fonte e executar um conjunto de analisadores estáticos, armazenando de maneira uniforme as análises feitas. Partindo dessa conclusão preliminar iniciamos a elaboração da arquitetura do kiskadee que será apresentada na próxima seção.

## 3.2 Kiskadee

kiskadee é um sistema de monitoramento e análise estática contínua de código fonte, implementado utilizando a linguagem de programação Python. Seu principal objetivo é monitorar o lançamento de novas versões e a cada nova versão executar um conjunto de analisadores no código fonte. Cada análise gerada será armazenada em um banco de dados relacional, viabilizando salvar diversas análises pertencentes à uma versão especí-

[4] https://gcc.gnu.org/
[5] https://clang-analyzer.llvm.org/
[6] http://findbugs.sourceforge.net/
[7] https://github.com/fedora-static-analysis/mock-with-analysis
[8] https://coala.io

fica de um projeto. Isso implica que em um intervalo de versões de um dado projeto, o kiskadee terá um conjunto de análises estáticas feitas pelos analisadores definidos pelo usuário, possibilitando uma análise da qualidade do projeto ao longo do tempo. Para que o monitoramento seja possível, o kiskadee precisa saber aonde procurar pelos projetos e suas versões. Isso é feito de duas maneiras: monitorando distribuições Linux e seus pacotes ou monitorando diretamente o *upstream*. A escolha por distribuições Linux se deu pela experiência dos envolvidos no desenvolvimento com tais distribuições, mas uma das características do kiskadee é ser genérico a ponto dessa escolha ser irrelevante, ou seja, devemos ser capazes de monitorar qualquer fonte pública de código fonte, desde que haja um padrão no processo de lançamento de novas versões. A Figura 3 apresenta a arquitetura do kiskadee e nas próximas subseções faremos uma revisão de cada componente que compõe a ferramenta.

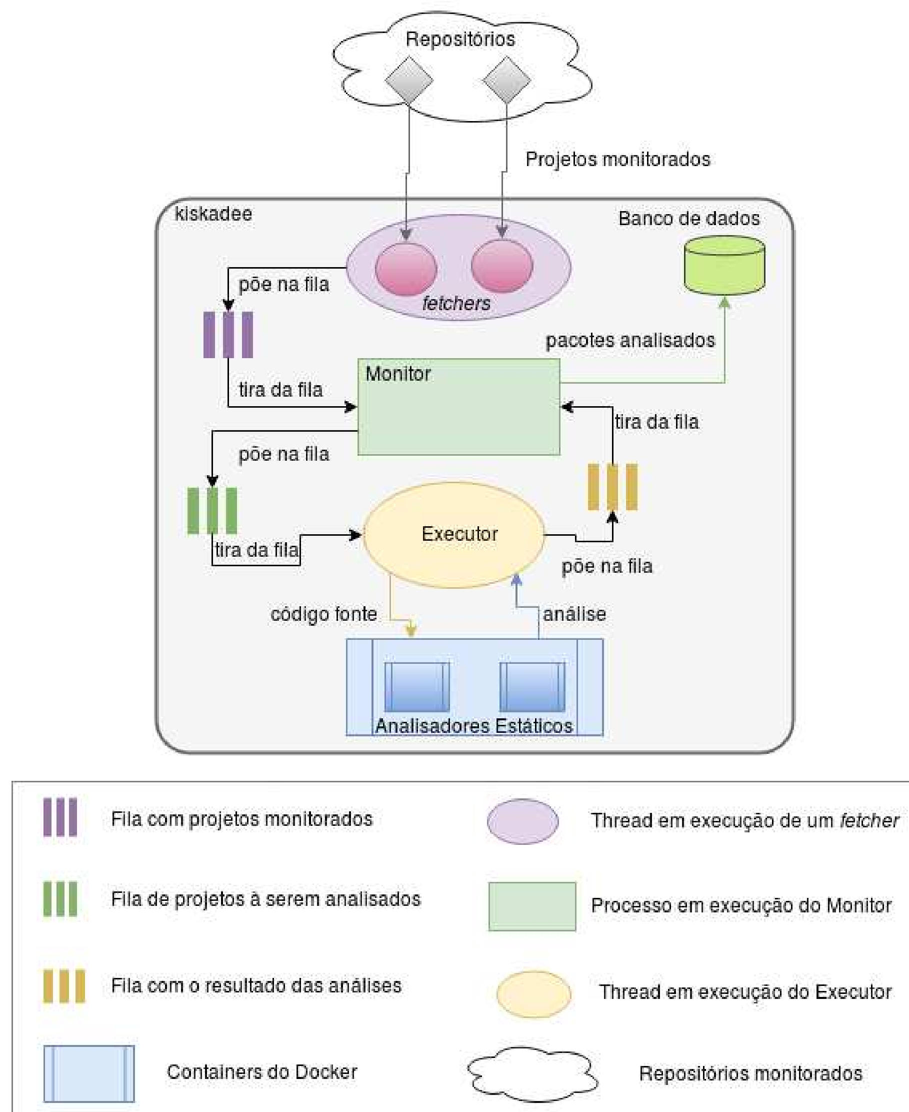



Figura 3 – Arquitetura do kiskadee

### 3.2.1 Repositórios

A camada de Repositórios apresentada na Figura 3 representa os projetos monitorados pelo kiskadee. Essa camada pode ser um *upstream*, uma distribuição Linux ou até mesmo um serviço que consiga informar dados mínimos sobre um código fonte que possa ser monitorado. A proposta de coleta contínua depende que nestes repositórios novas versões dos projetos monitorados sejam disponibilizadas. Durante o desenvolvimento do kiskadee percebemos que um repositório pode ser ativo ou passivo, de forma que, quando ativo o repositório notifica que uma nova versão de um projeto está disponível para ser analisada, e quando passivo é papel do kiskadee coletar dados do repositório para identificar novas versões. De qualquer maneira é papel do desenvolvedor que tem interesse em monitorar determinado repositório, estabelecer de que forma o kiskadee terá acesso ao código fonte que será analisado. É importante deixar claro que o núcleo da arquitetura está nos componentes Monitor e Executor, os *fetchers* e analisadores compõe a parte que pode ser customizada conforme a necessidade do desenvolvedor, ou seja, é possível adicionar tanto novos repositórios à serem monitorados quanto novas ferramentas de análise estática.

### 3.2.2 Fetchers

Existem diversas plataformas de desenvolvimento de software que proveem, além de outros serviços, uma maneira de disponibilizar versões de um projeto que podem ser utilizadas pelo usuário. Essas plataformas, em sua grande maioria, disponibilizam tais versões compactadas, normalmente em formato *zip* ou *tar.gz*. A diversidade de plataformas existentes que gostaríamos de monitorar e a decisão de não vincular as particularidades de cada uma ao núcleo do kiskadee, nos fez pensar uma arquitetura que permitisse monitorar diferentes plataformas sem que o desenvolvedor precisasse alterar diferentes partes do código da ferramenta. Essa decisão foi implementada por meio de pontos de extensão no código do kiskadee, chamados de *fetchers*. O mesmo contexto que envolve a diversidade de plataformas de desenvolvimento também se aplica a diversidade de distribuições Linux que existem, e monitorar diferentes distribuições também faz parte do nosso escopo.

Um *fetcher* para o kiskadee é uma classe Python que deve possuir três comportamentos esperados pelo restante da arquitetura, e que de modo geral são necessários para viabilizar o monitoramento. Os comportamentos são os seguintes:

- *Fetcher().watch()*: O método watch é utilizado pelo componente Monitor para inicializar cada *fetcher* definido como ativo pelo usuário. A implementação desse método fica a critério do desenvolvedor, mas de maneira geral ele é utilizado para monitorar o repositório desejado, identificando pacotes com potencial de serem analisados pelo kiskadee. Iniciar o *fetcher* de maneira não blocante (por meio de uma

*thread*), permite que o seu funcionamento ocorra de forma independente, tanto do núcleo do kiskadee quanto de outros *fetchers* que podem estar rodando. A comunicação entre os *fetchers* e o Monitor é feita utilizando a implementação padrão de fila do Python, permitindo que ambos interajam apenas no momento da criação das *threads*. Da inicialização em diante os dados que o Monitor utiliza são retirados diretamente da fila. Ao identificar um pacote no repositório monitorado, o *fetcher* enfileira esse pacote na fila compartilhada com o Monitor. Dizer que o *fetcher* enfileira um pacote significa que um dicionário Python é criado com informações relacionadas ao pacote. Esse dicionário pode ser visto na Listagem 3.1:

Listagem 3.1 – Exemplo de pacote enfileirado

```
{'name': 'foo',
 'version': '1.2.5-6',
 'fetcher': kiskadee.fetchers.debian
}
```

- *Fetcher().get_sources(package)*: Comportamento responsável por retornar um caminho na árvore de diretórios para o código fonte de um pacote. Esse comportamento acessa o repositório monitorado e baixa o código fonte compactado. Como a comunicação entre *fetcher* e Monitor é feito por meio de uma fila, esse método pode ser chamado a qualquer momento, pois um pacote que foi enfileirado não necessariamente vai ter o seu código fonte automaticamente baixado do repositório monitorado. Isso ocorre porque o que é de fato enfileirado são informações a respeito do pacote como, nome, versão e o *fetcher* que está monitorando tal repositório. Essas mesmas informações enfileiradas no método *watch* são passadas para o método *Fetcher().get_sources(package)*, para que ele possa fazer o *download* do código fonte quando solicitado.

- *Fetcher().compare_versions(new, old)*: Comportamento responsável por comparar duas versões de um pacote. É utilizado no momento de decidir se um pacote deve ser srepositório ou não, pois se a versão de um pacote já existe no banco de dados ela é ignorada e uma nova análise não é realizada. Decidimos mover esse comportamento para o *fetcher* pois nem todos os repositórios monitorados lançam novas versões seguindo um mesmo padrão semântico. Cabe ao *fetcher* informar ao kiskadee como comparar duas versões de um pacote que foi monitorado por ele.

Com esses comportamentos o ciclo de vida de um *fetcher* do kiskadee pode ser listado da seguinte forma:

- Continuamente verifica se há pacotes novos no repositório monitorado.

- Enfileira os pacotes novos na fila compartilhada com o Monitor.

- Quando o pacote enfileirado for processado (retirado da fila pelo componente Monitor), compara a versão do pacote a ser analisado com a última versão do banco de dados.
- Quando solicitado, faz o download do código fonte do pacote utilizando o argumento informado (o argumento terá o formato da Listagem 3.1).

### 3.2.3 Monitor

O componente Monitor é responsável por inicializar os *fetchers* do kiskadee e controlar quais pacotes devem ou não ser analisados pelo componente Executor. Os *fetchers* que serão inicializados pelo Monitor são definidos no arquivo de configuração *kiskadee.conf* e no apêndice D você pode visualizar um exemplo desse arquivo. Para saber se um pacote é realmente novo ou não, o componente compara a versão do pacote disponível no repositório, com a versão desse mesmo pacote no banco do kiskadee. Se a versão no repositório for maior que a versão presente no banco, o Monitor escreve uma mensagem em uma fila, informando que existe um novo pacote para ser analisado. Vale lembrar que essa comparação é feita invocando o método *Fetcher().compare_versions(new, old)* do *fetcher* responsável por enfileirar tal pacote. Ao reconhecer que uma versão de um pacote não existe no banco, o componente Monitor enfileira-a na fila compartilhada com o componente Executor. Quando o componente Executor encerra a análise dessa versão, encaminha de volta ao Monitor a análise realizada. O Monitor retira da fila tal análise, e só então salva o pacote analisado no banco.

### 3.2.4 Executor

O componente Executor é responsável por executar os diferentes analisadores, que pertencem a um *fetcher*, em um código fonte qualquer. Note que dentre as informações de um pacote que trafegam nas filas do kiskadee, uma delas é o módulo do *fetcher* que enfileirou tal pacote. Isso permite que o Executor chame o método *get_sources* no momento de rodar uma nova análise. Quais analisadores determinado *fetcher* utiliza são definidos no arquivo *kiskadee.conf*, e uma das responsabilidades do Executor é rodar análises utilizando apenas tais analisadores. Para executar os analisadores estáticos um container Docker[9] é inicializado e dentro dele a análise é feita. Cada analisador utilizado pelo kiskadee possui um *parser* no projeto Firehose, e logo após a análise ser feita, o respectivo *parser* é invocado. A análise que é salva no banco de dados é o resultado do *parser* e não o resultado direto da análise feita pelo analisador. Para incluir um novo analisador no kiskadee, um *parser*, com o mesmo nome do analisador, deve ser implementado no Firehose. Um exemplo prático disso ocorreu quando decidimos incluir a ferramenta Flawfinder como um

[9] https://www.docker.com/

analisador disponível no kiskadee. Não existia um *parser* para essa ferramenta no projeto Firehose, e para que o componente Executor pudesse realizar análises com o o Flawfinder tivemos que enviar um *patch* para o repositório oficial do Firehose[10]. Na versão 0.5 do Firehose o *parser* do Flawfinder foi incorporado ao código oficial do projeto e desde a versão 0.1 do kiskadee é possível realizar análises com o Flawfinder. Vale ressaltar que essa contribuição para o projeto Firehose foi a forma que definimos para tornar a proposta para o GSoC mais consistente, apresentando uma contribuição prática para um projeto que é central para o kiskadee.

### 3.2.5 Analisadores Estáticos

Os analisadores estáticos no kiskadee são executados por meio do componente Executor utilizando um container Docker. Para executar o analisador no diretório correto, o caminho retornado pelo método *Fetcher().get_sources(package)* é utilizado para criar um volume compartilhado ao inicializar o *container*.

A experiência inicial que tivemos, ao tentar utilizar o Debile para realizar análises estáticas no código fonte de pacotes Debian, nos mostrou que uma arquitetura complexa e pouco documentada pode inviabilizar o uso de qualquer ferramenta. O fato do Debile poder ser executado em diferentes contextos, mas ser altamente acoplado a infraestrutura e às ferramentas do Debian, fez com que demorássemos para conseguir gerar nossa primeira análise estática. Essa demora foi causada, principalmente, pela falta de documentação e por certas decisões arquiteturais que dificultavam o uso do Debile. Com o kiskadee, buscamos delimitar bem o seu escopo, simplificando a arquitetura dos módulos centrais (Monitor e Executor), e criando uma abordagem genérica para a adição de novos *fetchers* e analisadores, sem a necessidade de um estudo completo da ferramenta. Os próximos dois capítulos irão apresentar, respectivamente, de que maneira foi possível alcançar tal abordagem e os resultados obtidos pela arquitetura apresentada neste capítulo.

[10] https://github.com/fedora-static-analysis/firehose

# 4 Implementação

Como resultado deste trabalho, chegamos a um conjunto de funcionalidades que nos permitiram monitorar alguns repositórios e rodar análises estáticas para diversos projetos. Todo o desenvolvimento do kiskadee está sendo feito na plataforma de desenvolvimento *Pagure*, e o repositório pode ser acessado por meio do endereço <https://pagure.io/kiskadee>. É nesse repositório que fazemos toda a gestão do desenvolvimento, incluindo o lançamento de novas versões da ferramenta. Um ambiente de integração contínua utilizando a ferramenta Jenkins[1] também foi definido, e o endereço para o ambiente está no *README.md* do repositório. Esse ambiente é importante para assegurar que todo código que entre na branch *master* do repositório mantenha o comportamento da ferramenta estável, e no contexto do kiskadee isso só é possível por meio de testes automatizados. Estamos utilizando a biblioteca *Sphinx*[2] para realizar a documentação do código fonte, e todas as instruções para gerar tal documentação e executar o kiskadee podem ser encontradas no repositório. Nas próximas seções faremos uma revisão do que foi implementado na versão 0.2.2 do kiskadee, apresentando alguns detalhes de implementação quando julgarmos necessário.

## 4.1 Fetchers

Como foi falado no Capitulo 3, todo *fetcher* do kiskadee possui certos comportamentos esperados pelo restante da arquitetura, de modo que tanto para o componente Monitor quanto para o Executor é irrelevante a maneira como os comportamentos são implementados internamente porcada *fetcher*. O que importa é que ao informar os parâmetros esperados, o *fetcher* retorne um dado que, tanto o componente Monitor quanto o componente Executor possam utilizar para prosseguir com suas responsabilidades. Para que todo *fetcher* no kiskadee implemente tais comportamentos, o conceito de *polimorfismo* foi utilizado, sendo implementado por meio de uma classe abstrata que define quais os comportamentos todo *fetcher* do kiskadee deve possuir. Ao herdar dessa classe abstrata e sobrescrever os métodos herdados, permite-se que os componentes Monitor e Executor possam invocar um mesmo método, independente do *fetcher* que está envolvido na operação. Utilizar o conceito de polimorfismo nos permitiu adicionar novos *fetchers* sem ter que alterar o restante da aplicação, o que facilita tanto a manutenção quanto a evolução do kiskadee. A Listagem 4.1 demonstra como essa classe abstrata foi implementada.

Listagem 4.1 – Definição dos comportamentos de um extitfetcher

1 https://jenkins.io/
2 http://www.sphinx-doc.org/en/stable/

```
class Fetcher():

    __metaclass__ = abc.ABCMeta

    def __init__(self):
        full_name = inspect.getmodule(self).__name__
        self.name = full_name.split('.')[-1]
        config_section = self.name + '_ extit{fetcher}'
        self.config = kiskadee.config[config_section]

    @abc.abstractmethod
    def get_sources(package):
        raise NotImplementedError('get_sources must be defined by
           fetcher')

    @abc.abstractmethod
    def watch(self):
        raise NotImplementedError('watch must be defined by fetcher')

    @abc.abstractmethod
    def compare_versions(self, new, old):
        raise NotImplementedError('compare_versions must be defined by
           fetcher')
```

Note que a classe abstrata não define a implementação de cada método, isso fica a critério de cada *fetcher*. O que ela define é a interface de cada método que deverá ser implementado, para que tanto o componente Monitor quanto Executor possam funcionar corretamente.

Todos os *fetchers* do kiskadee ficam dentro do pacote Python *Fetcher*. Cada um deles, se definidos como ativos, são carregados de maneira dinâmica no momento em que o kiskadee for inicializado pelo usuário. É por meio do arquivo de configuração que definimos quais deles devem ser inicializados. No apêndice D é possível ver um exemplo dessa definição. Além do componente Executor, cada *fetcher* do kiskadee é executado como uma *thread*, permitindo que as várias partes da ferramenta funcionem de maneira independente, se comunicando por meio da implementação de filas do Python. Existem três filas que são utilizadas dentro da aplicação. A primeira é a fila de pacotes, utilizada pelos *fechers* para enfileirar pacotes que devem ser analisados pelo componente Executor. A segunda é a de analises, que é utilizada pelo Monitor para enviar ao Executor os pacotes que realmente precisam ser analisados. A terceira é a fila de resultados, utilizada pelo Monitor para receber do Executor as análises feitas. Na arquitetura atual do kiskadee o componente Monitor tem papel central, pois é ele que controla quais pacotes devem ser enviados para a análise, e srepositórios no banco de dados. Para que a comunicação entre os *fetchers* e o componente Monitor seja possível, tiramos proveito do fato de que uma

*thread* compartilha variáveis globais definidas a nível de processo. Quando inicializamos a implementação de filas do Python antes de inicializarmos os *fetchers*, viabilizamos a comunicação entre os diversos componentes que precisam se comunicar e conseguimos manter as várias partes do kiskadee desacopladas umas das outras. A inicialização tanto dos *fetchers* quanto do componente Executor é feita pelo componente Monitor. A Listagem 4.2 apresenta como isso é feito atualmente.

Listagem 4.2 – Inicialização dos Fetcher do kiskadee

```
def initialize(self):
    _start(self.monitor)
    fetchers = kiskadee.load_fetchers()
    for fetcher in fetchers:
        self._save_ fetcher(fetcher)
        _start(fetcher.Fetcher().watch)
        time.sleep(1)
    _start(kiskadee.runner.runner, True)


def _start(module, joinable=False, timeout=None):
    module_as_a_thread = threading.Thread(target=module)
    module_as_a_thread.daemon = True
    module_as_a_thread.start()
    if joinable or timeout:
        module_as_a_thread.join(timeout)
```

### 4.1.1 Debian

Na versão 0.2 do kiskadee, três *fetchers* estão disponíveis para uso. O primeiro é responsável por monitorar o repositório FTP (File Transfer Protocol) da distribuição Linux Debian, utilizando os pacotes disponibilizados na versão *sid* da distribuição. Essa versão, também chamada de não estável, é o ponto de entrada de novos pacotes no Debian. Monitorar a versão sid permite avaliar pacotes com versões próximas às que são lançadas pelos *upstream*, de forma que as análises feitas pelo kiskadee podem auxiliar os desenvolvedores a aplicarem correções no código fonte de pacotes considerados não estáveis. Independente da distribuição que estivermos monitorando, é importante sempre analisar pacotes que estejam em uma versão de desenvolvimento, como é o caso do Debian sid.

Para monitorar o repositório, a cada intervalo de tempo definido pela variável *schedule* presente no arquivo *kiskadee.conf*, fazemos o download do arquivo *Sources.gz*, e a partir dele identificamos todos os pacotes disponíveis no repositório. Além de listar todos os pacotes presentes no repositório o Sources.gz também disponibiliza outras informações relevantes sobre cada pacote, incluindo o caminho para o código fonte empacotado. A partir dessas informações o *fetcher* enfileira todos os pacotes na fila utilizada pelo Monitor,

de modo que ele possa decidir quais pacotes devem ser analisados. Utilizando o nome e versão do pacote, como o apresentado na Listagem 3.1, o *fetcher* é capaz de fazer o download do código fonte e retornar o caminho para o arquivo compactado. É importante deixar claro que o kiskadee é executado na distribuição Linux Fedora, e para realizar o download do código fonte de um pacote Debian utilizamos uma ferramenta chamada *dget*[3]. No Debian, cada pacote existente na distribuição possui um arquivo que o descreve. Esse arquivo é composto pelo nome do pacote, sua versão e possui extensão *dsc*. Então um pacote chamado *foo*, com uma versão *1.0.0*, terá um arquivo dsc *foo-1.0.0.dsc*. Para fazer o download do código fonte do pacote foo precisamos informar para a ferramenta dget o endereço do arquivo dsc do pacote, presente no respositório Debian monitorado. No *fetcher* esse endereço é construido por meio do método *_dsc_url* que pode ser visto na Listagem 4.3.

Listagem 4.3 – Url passada como parâmetro para a ferramenta dget

```
def _dsc_url(self, source_data):
    """Build dsc url to download a debian package sources.

    This url is required by dget. e.g.:
    dget http://ftp.debian.org/debian/pool/main/0/0ad/0ad_0.0.21-2.
        dsc

    """
    name = source_data['name']
    version = source_data['version']
    directory = source_data['meta']['directory']
    return ''.join([self.config['target'], '/',
                    directory, '/', name, '_', version, '.dsc'])
```

No exemplo do nosso pacote foo, o retorno do método *_dsc_url* seria a url <http://ftp.debian.org/debian/pool/main/f/foo/foo_1.0.0.dsc>. A partir dessa url a ferramenta dget é capaz de fazer o download do código fonte do pacote, quando tal ação for solicitada pelo componente Executor. Note que várias informações necessárias para montar o endereço esperado pelo dget são retirados do arquivo Sources.gz, e por isso a sua importância para o *fetcher*. Um exemplo de um pacote descrito pelo arquivo Sources.gz pode ser visto no apendice D.

### 4.1.2 Anitya

O segundo *fetcher* é responsável por monitorar o serviço *Anitya*[4]. Anitya é um sistema de monitoramento de versões de projetos, que monitora o lançamento de novas

[3] O dget está empacotado no projeto Fedora por meio do pacote devscripts
[4] https://release-monitoring.org/

versões e os publica no *fedmsg*[5]. O Anitya monitora diferentes *upstream*, e se olharmos o código fonte do sistema é possível perceber que a forma como esse monitoramento é feito se assemelha com os *fetchers* do kiskadee. Cada *upstream* monitorado é um modulo interno do Anitya que precisou ser implementado por alguém. Alguns dos *upstream* monitorados pelo Anitya podem ser vistos a seguir, e a lista completa pode ser vista no link <https://release-monitoring.org/about>:

- cpan.py para projetos pearl hospedado em <http://www.cpan.org/>
- debian.py para projetos hospedado em <http://ftp.debian.org/debian/pool/main/>
- github.py para projetos hospedado em <github.com>
- rubygems.py para projetos hospedado em <rubygems.org>

Para que um projeto disponibilizado em um desses *upstream* possa ser monitorado pelo Anitya, tal projeto tem de ser cadastrado na plataforma, informando o nome do projeto, o prefixo utilizado no lançamento de novas versões e qual modulo interno do Anitya será usado para realizar o monitoramento. Quanto mais projetos forem cadastrados no Anitya, mais projetos poderão ser analisados pelo kiskadee. O Anitya possui um modulo interno para monitorar a distribuição Debian, mas não são todos os pacotes disponibilizados na distribuição que estão cadastrados no Anitya. É fato que se a cobertura de pacotes do Debian feita pelo Anitya chegar a um nível considerado bom pela comunidade do kiskadee, o *fetcher* do Debian pode vir a ser tornar obsoleto em algum momento. É importante dizer que o Anitya diminui a complexidade de monitorar diferentes *upstream*, uma vez que é ele quem faz o trabalho de identificar quando novas versões dos projetos cadastrados são lançadas, diferente por exemplo do *fetcher* do Debian, que precisa estar continuamente processando o arquivo Sources.gz para identificar quais novos projetos foram disponibilizados na distribuição.

O fedmsg é um sistema de mensagens utilizado pelo projeto Fedora para integrar de maneira distribuída diversos serviços que precisam se comunicar de maneira desacoplada. Toda vez que um projeto monitorado pelo Anitya lança uma nova versão, uma mensagem representando tal evento é publicada no fedmsg. Todos os serviços que publicam eventos no fedmsg utilizam um formato JSON para as mensagens, Nesse link <https://fedora-fedmsg.readthedocs.io/en/latest/> é possível visualizar os serviços e os tópicos utilizados para a publicar mensagens no fedmsg.

Os eventos monitorados pelo Anitya são diversos, mas no contexto do kiskadee o único evento que nos interessa é o *version.update*, que representa o momento em que um *upstream* monitorado libera uma nova versão. Capturar eventos relacionados apenas

[5] http://www.fedmsg.com/en/latest/

ao lançamento de novas versões é simples, pois por utilizar o fedmsg como serviço de mensagens, o kiskadee assina apenas o tópico em que eventos de novas versões identificadas pelo Anitya são publicados.

Para receber os eventos do Anitya publicados no fedmsg o kiskadee interage, também por meio de mensagens, com um *daemon* chamado *fedmsg-hub*. A Figura 4 demonstra de maneira simplificada esse funcionamento. Para a comunicação entre o fedmsg-hub e o *fetcher* do Anitya, estamos utilizando uma biblioteca chamada ZeroMQ[6] que, assim como o fedmsg, funciona como um sistema distribuído de mensagens. Dessa forma o fedmsg-hub publica as mensagens que ele recebe em um tópico do servidor ZeroMQ, e o kiskadee assina esse tópico para receber as mensagens. Isso nos permitiu desacoplar da implementação do *fetcher* qualquer a interação com os serviços do fedmsg.

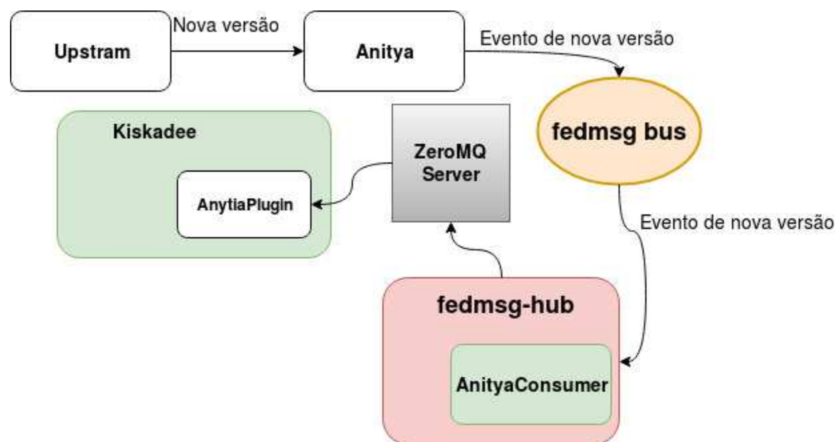



Figura 4 – Comunicação com o Anitya

### 4.1.3 Juliet

O terceiro *fetcher* é responsável por analisar a suite de testes *Juliet*[7]. É um *fetcher* simples que basicamente baixa os casos de teste disponibilizados pelo projeto e roda os analisadores estáticos definidos no arquivo kiskadee.conf. De qualquer forma todos esses *fetchers* implementam os comportamentos definidos na classe base de *Fetcher*, já comentada anteriormente.

## 4.2 Analisadores Estáticos

O responsável por executar os analisadores estáticos em um código fonte qualquer é o componente Executor. Vimos que o componente Monitor enfileira pacotes que devem ser analisados, e o Executor consome essa fila para executar as análises. Cada pacote analisado está relacionado a um *fetcher*, e cada *fetcher* define quais os analisadores serão

[6] http://zeromq.org/
[7] https://samate.nist.gov/SRD/testsuite.php

utilizados no momento da análise. Assegurar que as análises sejam feitas corretamente, e salvar o resultado dessas análises no banco de dados é papel do componente Executor. Como todo *fetcher* do kiskadee implementa os comportamentos descritos na Listagem 4.1, o Executor é capaz de chamar o método *Fetcher().get_sources(package)*, responsável por retornar o caminho para o código fonte a ser analisado, independente de qual *fetcher* monitorou o pacote em questão. Para executar uma análise os analisadores são chamados sequencialmente, informando o caminho para o código fonte a ser analisado. A execução do analisador é feito utilizando a plataforma Docker, a partir de uma biblioteca Python que faz a interface entre o kiskadee e as chamadas necessárias para realizar a analise dentro do *container*. O trecho 4.4 demonstra um exemplo de Dockerfile, utilizado para executar uma análise estática utilizando a ferramenta Flawfinder. Para que um *container* possa ser criado no momento de realizar a análise, as imagens Docker devem ser geradas antes de inicializar o kiskadee.

Listagem 4.4 – Exemplo de Dockerfile

```
FROM fedora

RUN dnf -y update && dnf clean all
RUN dnf -y install python flawfinder && dnf clean all

ENTRYPOINT ["flawfinder"]
```

Na versão 0.2, o kiskadee disponibiliza os seguintes analisadores:

- cppcheck, versão 1.79
- flawfinder, versão 1.31
- clang-analyzer, versão 3.9.1
- frama-c, versão 1.fc25

## 4.3 Modelo de domínio

A versão 0.1 do kiskadee trouxe um conjunto mínimo de funcionalidades que nos permitiram monitorar alguns repositórios e analisar alguns pacotes, servindo mais como uma prova de conceito do que como um sistema de monitoramento real. Uma das limitações dessa versão era que não era possível salvar diferentes análises, feitas por diferentes analisadores, em uma mesma versão de um pacote. A versão 0.2 trouxe mudanças ao modelo de domínio do kiskadee, de forma que agora é possível salvar diferentes análises para uma mesma versão. A Figura 5 apresenta como está estruturado o modelo de domínio do kiskadee na versão 0.2. Vale ressaltar que estamos utilizando o projeto *sqlachemy*[8] para

[8] https://www.sqlalchemy.org/

salvar o modelo de domínio no banco de dados. O sqlachemy é um Object Relational Mapper (ORM), ou seja, ele mapeia o modelo de domínio em tabelas no banco de dados, permitindo utilizar padrões de persistência, ao invés de código SQL puro.

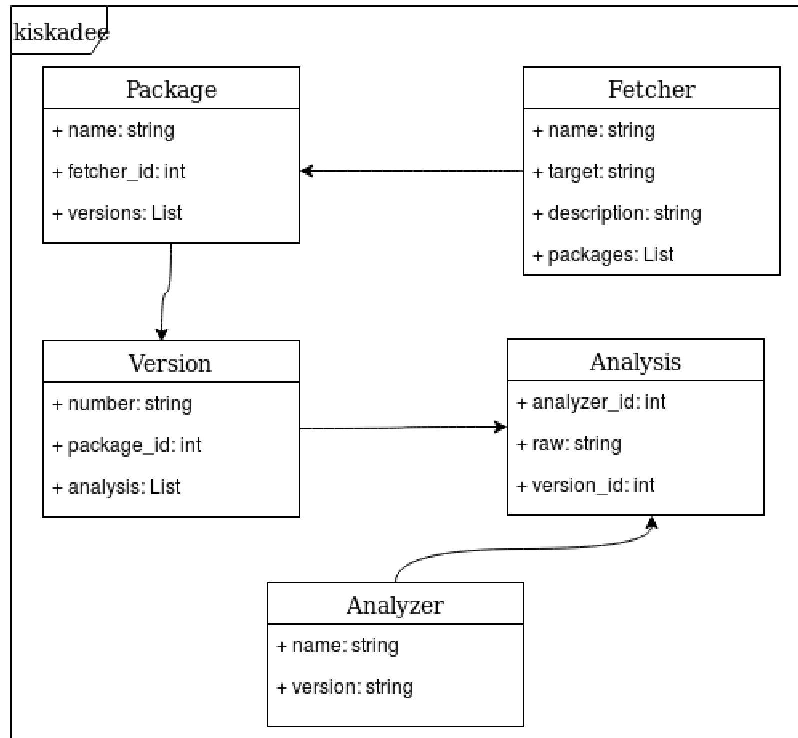



Figura 5 – Modelo de domínio do kiskadee

O modelo *Package*, representa um projeto que foi monitorado e analisado pelo kiskadee. O responsável por esse monitoramento é representado pelo modelo *Fetcher*, que está relacionado a vários *Packages*. Um *Package* possui várias *Versions*, e cada uma terá uma análise correlacionada. Uma análise é sempre feita por um *Analyzer*.

# 5 Exemplo de uso

Neste capítulo apresentaremos os resultados obtidos com a implementação descrita no Capitulo 4, e algumas conclusões que percebemos ao longo do desenvolvimento do kiskadee. Inicialmente vamos definir os dois exemplos em que o monitoramento do kiskadee pode atuar. O primeiro exemplo, demonstrado pelo fluxograma 6, ocorre quando o kiskadee monitora distribuições Linux. Nesse exemplo um *upstream* libera uma nova versão do projeto, essa versão é empacotada para as distribuições, o sistema de monitoramento do kiskadee percebe que existe uma nova versão de um pacote, baixa o código fonte, realiza as análises e às salva no banco de dados. Nesse exemplo o ganho é maior para a distribuição, pois todos os pacotes novos que forem lançados serão monitorados, mas não é interessante para o *upstream*, pois nem todas as novas versões serão empacotadas e por consequência não serão analisadas.

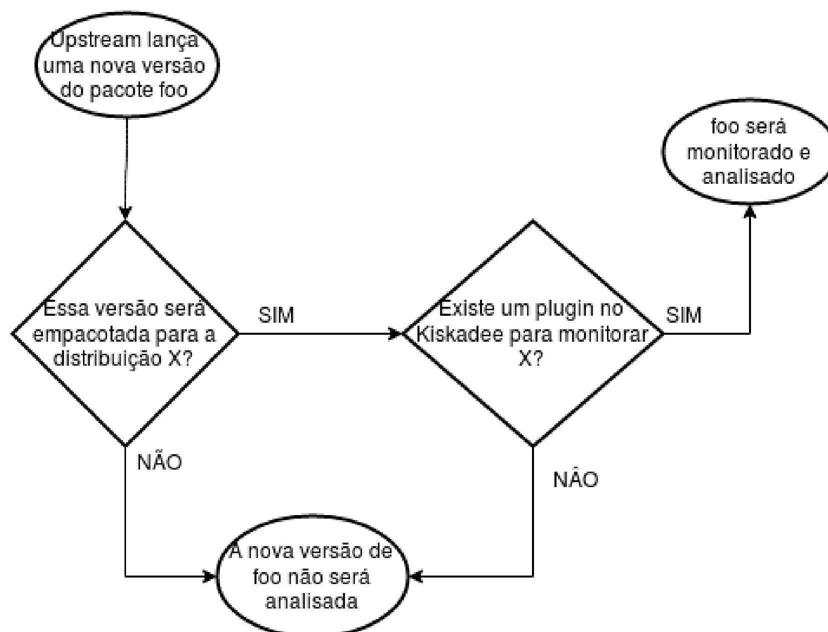



Figura 6 – Fluxo de monitoramento de uma distribuição

Uma primeira implicação desse exemplo de uso é que teremos que implementar um *fetcher* para cada distribuição que quisermos monitorar, fato que aumenta o consumo de recursos da ferramenta, e a complexidade de gerenciar tantas *thread* sendo executadas de maneira concorrente. Sabemos que conforme o número de repositórios monitorados pela ferramenta aumentarem, mudanças na arquitetura atual terão de ser feitas. A forma como executamos os *fetcher* é simples e funciona como o esperado, mas teremos que utilizar uma abordagem assíncrona mais robusta caso o número de *fetchers* aumente.

O segundo exemplo de uso, apresentado pelo Fluxograma 7, ocorre quando o

kiskadee, utilizando o projeto Anitya, monitora diretamente o lançamento de novas versões por parte do *upstream*. Esse é o exemplo de uso ideal, pois ao analisarmos o *upstream*, também analisaremos o código que foi, ou será, empacotado pela distribuição. É importante deixar claro que, caso iniciativas como o projeto Anitya não existissem, teríamos o mesmo problema de monitorar diferentes distribuições Linux, ou seja, teríamos que implementar um *fetcher* para cada plataforma de desenvolvimento (GitHub, GitLab[1], Bitbucket[2] e etc) que quiséssemos monitorar, implementando no kiskadee o serviço disponibilizado pelo projeto Anitya.

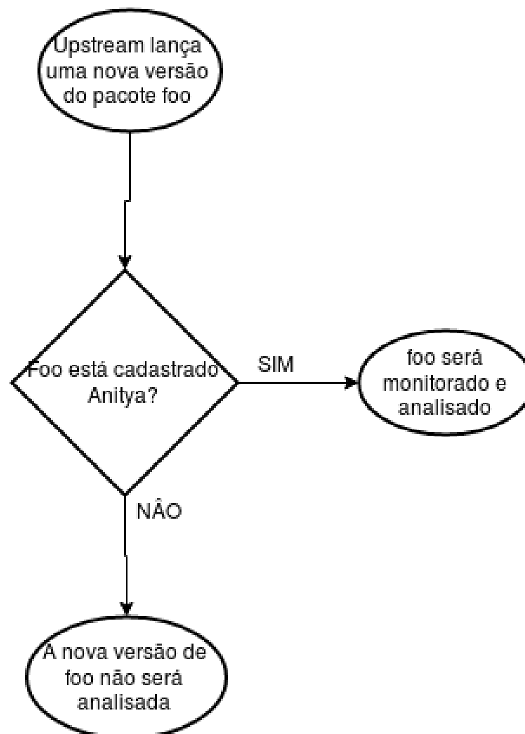
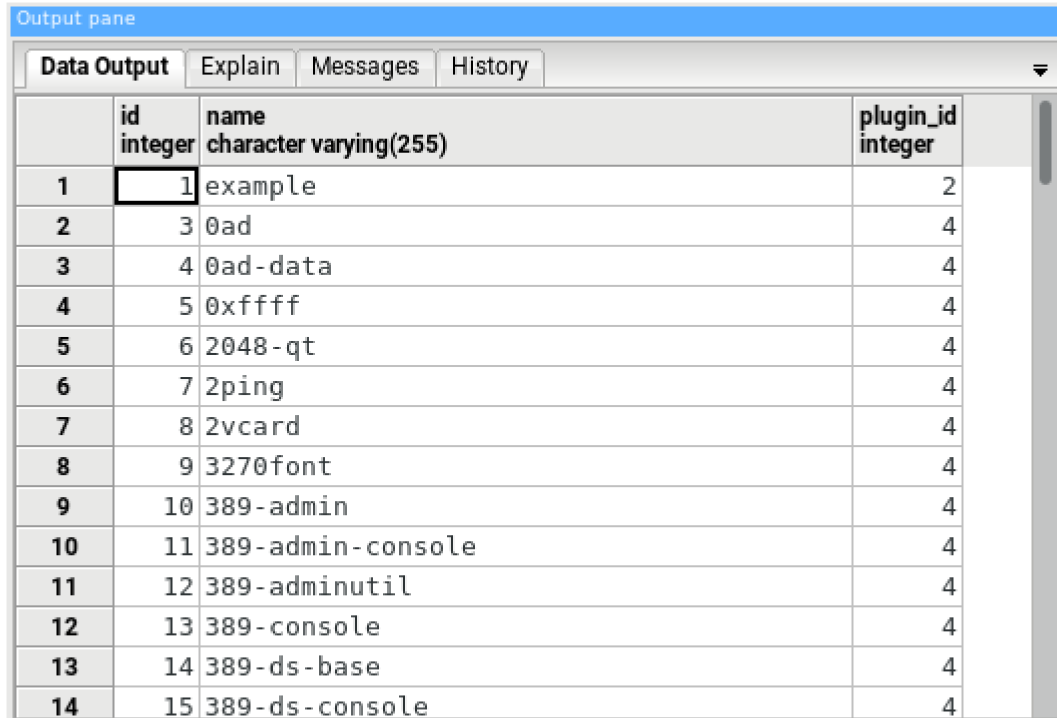



Figura 7 – Fluxo de Monitoramento de um upstream

Partindo dessa diferenciação entre monitorar distribuições e monitorar diretamente o *upstream* apresentaremos algumas análises feitas pelo kiskadee em projetos diversos. A partir do banco de dados do kiskadee, foi possível recuperar algumas análises feitas tanto com a ferramenta Cppcheck quanto com a ferramenta Flawfinder. Os projetos apresentados foram monitorados pelo extitfetcher do Debian e pelo extitfetcher do Anitya.

A Figura 8 apresenta alguns dos pacotes srepositórios no banco do kiskadee. Na versão em que coletamos os dados apresentados, 9713 pacotes foram srepositórios no banco de dados do kiskadee. Esse número de pacotes ocorre principalmente devido ao *fetcher* do Debian, que em um primeiro acesso ao repositório, irá salvar todos os pacotes descritos pelo Sources.gz.

[1] https://about.gitlab.com/
[2] https://bitbucket.org/



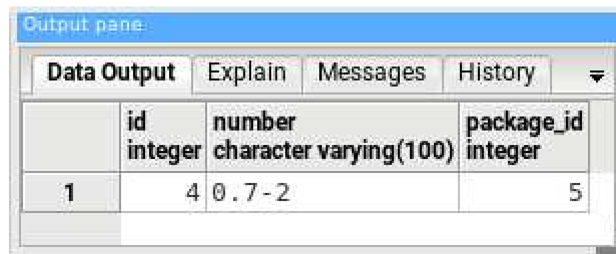
Output pane

Data Output | Explain | Messages | History

| | id integer | name character varying(255) | plugin_id integer |
|---|---|---|---|
| 1 | 1 | example | 2 |
| 2 | 3 | 0ad | 4 |
| 3 | 4 | 0ad-data | 4 |
| 4 | 5 | 0xffff | 4 |
| 5 | 6 | 2048-qt | 4 |
| 6 | 7 | 2ping | 4 |
| 7 | 8 | 2vcard | 4 |
| 8 | 9 | 3270font | 4 |
| 9 | 10 | 389-admin | 4 |
| 10 | 11 | 389-admin-console | 4 |
| 11 | 12 | 389-adminutil | 4 |
| 12 | 13 | 389-console | 4 |
| 13 | 14 | 389-ds-base | 4 |
| 14 | 15 | 389-ds-console | 4 |

Figura 8 – Lista de pacotes srepositórios pelo kiskadee

Iremos então selecionar o pacote *0xffff*, para visualizarmos de que maneira uma análise está organizada no banco de dados. A Figura 9 apresenta qual a versão mais recente desse pacote no banco.



Output pane

Data Output | Explain | Messages | History

| | id integer | number character varying(100) | package_id integer |
|---|---|---|---|
| 1 | 4 | 0.7-2 | 5 |

Figura 9 – Versão analisada do pacote 0xffff

Antes de visualizarmos a análise feita para o pacote 0xffff na versão monitorada é importante pontuarmos que, dado os analisadores estáticos presentes hoje no kiskadee, apenas projetos escritos nas linguagens C/C++ serão analisados. Isso ocorre devido a uma limitação de escopo definida por nós. Felizmente a arquitetura do kiskadee pode ser evoluída para suportar analisadores de outras linguagens. O fato de analisarmos apenas projetos C/C++ faz com que o kiskadee tenha que ignorar projetos que não sejam implementados nessas linguagens, ou seja, existirão pacotes no banco de dados que não terão uma análise correlacionada.

Seguindo a visualização dos dados relacionados ao pacote 0xffff, as Figuras 10 e 11, apresentam duas análises feitas pelo kiskadee, para a versão 0.7-2 do pacote. A primeira, feita pela ferramenta Cppcheck, e a segunda pela ferramenta Flawfinder. O pacote 0xffff

foi monitorado e analisado pelo *fetcher* do Debian. Caso, na versão sid da distribuição, uma nova versão do pacote seja lançada, uma nova análise será feita pelo kiskadee.


```
- <analysis>
  - <metadata>
      <generator name="cppcheck" version="1.79"/>
    </metadata>
  - <results>
    - <issue severity="style" test-id="variableScope">
      - <message>
          The scope of the variable \'value\' can be reduced.
        </message>
      - <notes>
          The scope of the variable \'value\' can be reduced. Warning: Be careful when fixing this
          cppcheck will write that the scope for \'i\' can be reduced:\\012void f(int x)\\012{\\012
          n < 10; ++n) {\\012 // it is possible but not safe to move \'int i = 0;\' here\\012 do_som
          reduce the variable scope 1 level.
        </notes>
      - <location>
          <file given-path="/src/0xffff-0.7/src/cal.c"/>
          <point column="0" line="170"/>
        </location>
      </issue>
    - <issue severity="information" test-id="ConfigurationNotChecked">
      - <message>
          Skipping configuration \'SSIZE_MAX\' since the value of \'SSIZE_MAX\' is unknown. U
        </message>
      - <location>
          <file given-path="/src/0xffff-0.7/src/cal.c"/>
          <point column="0" line="102"/>
        </location>
      </issue>
    - <issue severity="style" test-id="variableScope">
        <message>The scope of the variable \'crc\' can be reduced.</message>
```


Figura 10 – análise do pacote 0xffff, feita pelo Cppcheck


```
- <analysis>
  - <metadata>
      <generator name="flawfinder" version="1.31"/>
    </metadata>
  - <results>
    - <issue cwe="120" severity="4" test-id="buffer">
      - <message>
          Does not check for buffer overflows when copying to destination (CWE-120). Consider using strcpy_s, s
        </message>
      - <location>
          <file given-path="/src/0xffff-0.7/src/fiasco.c"/>
          <point column="0" line="194"/>
        </location>
      </issue>
    - <issue cwe="120" severity="4" test-id="buffer">
      - <message>
          Does not check for buffer overflows (CWE-120). Use sprintf_s, snprintf, or vsnprintf.
        </message>
      - <location>
          <file given-path="/src/0xffff-0.7/src/fiasco.c"/>
          <point column="0" line="513"/>
        </location>
      </issue>
    - <issue cwe="134" severity="4" test-id="format">
      - <message>
          If format strings can be influenced by an attacker, they can be exploited (CWE-134). Use a constant for
        </message>
      - <location>
          <file given-path="/src/0xffff-0.7/src/global.h"/>
          <point column="0" line="14"/>
        </location>
      </issue>
```


Figura 11 – análise do pacote 0xffff, feita pelo Flawfinder

Apresentaremos então, uma análise feita pelo *fetcher* que monitora o serviço Anitya. O pacote *Xe* foi analisado pelo kiskadee, ou seja, o serviço do Anitya publicou um evento no fedmsg informando uma nova versão do projeto, o fedmsg-hub recebeu esse evento de atualização, publicou uma mensagem no servidor do ZeroMQ, o kiskadee recebeu essa mensagem e executou a análise. Podemos ver pela Figura 12, que uma versão do projeto foi disponibilizada dois dias atrás.

Figura 12 – Nova versão do projeto Xe

Como esse projeto está cadastrado no Anitya, a Figura 13 demonstra o evento de atualização publicado no fedmsg e a Figura 14 apresenta parte do JSON publicado.

Figura 13 – Evento de mudança de versão


```
{
  "source_name": "datanommer",
  "certificate": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1J
  "i": 6,
  "timestamp": 1500337612.0,
  "msg_id": "2017-4089addf-605a-430b-b3ca-4d42f6ee3a89",
  "topic": "org.release-monitoring.prod.anitya.project.vers
  "source_version": "0.7.0",
  "signature": "EG3WNLG/B+1EJOUqfZNGyrmI1scCKPejWYAIT9Siv6
  "msg": {
    "project": {
      "regex": null,
      "name": "xe",
      "versions": [
        "v0.9",
        "v0.8",
        "v0.7.0",
        "v0.6.1"
      ],
```


Figura 14 – Trecho da mensagem publicada no fedmsg

Output pane

Data Output | Explain | Messages | History

| | id integer | name character varying(255) | plugin_id integer |
|---|---|---|---|
| 1 | 9761 | xe | 1 |

(a) Projeto Xe srepositório no banco

Output pane

Data Output | Explain | Messages | History

| | id integer | number character varying(100) | package_id integer |
|---|---|---|---|
| 1 | 10132 | v0.9 | 9761 |

(b) Versão analisada do projeto Xe

Figura 15 – Pacote e Versão do projeto Xe

Com o evento publicado no fedmsg e o fedmsg-hub informando o kiskadee de tal evento, a Figura 15 apresenta o projeto Xe srepositório no banco de dados e qual a versão analisada. Essa versão é a mesma lançada pelo *upstream* e monitorada pelo Anitya.

Como resultado do monitoramento, as Figura 16 e 17, apresentam duas análises realizadas no código fonte do projeto Xe, na versão v0.9. A primeira analise foi realizada com a ferramenta Cppcheck e a segunda com a ferramenta Flawfinder.

```
- <analysis>
  - <metadata>
      <generator name="cppcheck" version="1.79"/>
    </metadata>
  - <results>
    - <issue severity="style" test-id="unsignedLessThanZero">
      - <message>
          Checking if unsigned variable \'argmax\' is less than zero.
        </message>
      - <notes>
          The unsigned variable \'argmax\' will never be negative so it is either pointless or an error to check if it is.
        </notes>
      - <location>
          <file given-path="/src/xe-0.9/xe.c"/>
          <point column="0" line="379"/>
        </location>
      </issue>
    - <issue severity="error" test-id="memleakOnRealloc">
      - <message>
          Common realloc mistake: \'buf\' nulled but not freed upon failure
        </message>
      - <location>
          <file given-path="/src/xe-0.9/xe.c"/>
          <point column="0" line="197"/>
        </location>
      </issue>
    - <issue severity="information" test-id="ConfigurationNotChecked">
      - <message>
          Skipping configuration \'_SC_NPROCESSORS_ONLN\' since the value of \'_SC_NPROCESSORS_ONLN\' is unknown.
        </message>
      - <location>
          <file given-path="/src/xe-0.9/xe.c"/>
          <point column="0" line="326"/>
        </location>
      </issue>
    - <issue severity="information" test-id="ConfigurationNotChecked">
```

Figura 16 – Análise feita no projeto Xe, pelo Cppcheck

Todas as análises aqui apresentadas foram coletadas diretamente do banco de da-

```
- <analysis>
  - <metadata>
      <generator name="flawfinder" version="1.31"/>
    </metadata>
  - <results>
    - <issue cwe="78" severity="4" test-id="shell">
      - <message>
          This causes a new program to execute and is difficult to use safely (CWE-78). try using a library c
        </message>
      - <location>
          <file given-path="/src/xe-0.9/xe.c"/>
          <point column="0" line="255"/>
        </location>
      </issue>
    - <issue cwe="120" severity="3" test-id="buffer">
      - <message>
          Some older implementations do not protect against internal buffer overflows (CWE-120, CWE-20).
        </message>
      - <location>
          <file given-path="/src/xe-0.9/xe.c"/>
          <point column="0" line="384"/>
        </location>
      </issue>
    - <issue cwe="120" severity="2" test-id="buffer">
      - <message>
          Does not check for buffer overflows when copying to destination (CWE-120). Make sure destinatio
        </message>
      - <location>
          <file given-path="/src/xe-0.9/xe.c"/>
          <point column="0" line="202"/>
        </location>
      </issue>
    - <issue cwe="119" severity="2" test-id="buffer">
      - <message>
          Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues
          maximum possible length.
        </message>
      - <location>
          <file given-path="/src/xe-0.9/xe.c"/>
          <point column="0" line="235"/>
        </location>
```


Figura 17 – Análise feita no projeto Xe, pelo Flawfinder

dos do kiskadee. Atualmente a ferramenta está rodando em uma máquina virtual disponibilizada pela Universidade de São Paulo, e nessa versão que está em produção a arquitetura aqui apresentada é a que está sendo utilizada para realizar o monitoramento. A ferramenta utilizada para apresentar os dados srepositórios no banco foi o pgAdminIII[3], uma vez que estamos utilizando o banco de dados postgresql, tanto no ambiente de produção quanto no ambiente de desenvolvimento.

[3] https://www.pgadmin.org/

# 6 Considerações Finais

O processo de analisar código fonte por meio de ferramentas de análise estática, pode guiar o desenvolvedor em uma tarefa de refatoração do código, buscando diminuir fraquezas que possam levar a comportamentos inesperados ou até mesmo a vulnerabilidades exploráveis. Dessa forma, vamos retomar a questão-problema que motivou este trabalho:

*Como definir um sistema extensível para o monitoramento e análise estática contínua de código fonte?*

O caráter extensível do sistema que surgiu a partir da questão-problema deste trabalho, foi o que guiou boa parte das decisões arquiteturais que tomamos. Permitir que qualquer desenvolvedor, interessando em mensurar a qualidade de um projeto, consiga implementar um *fetcher*, que colete informações como nome, versão e aonde tal projeto está hospedado, permite atingir parte da característica extensível levantada pela questão-problema. Permitir que qualquer desenvolvedor, inclua uma nova ferramenta de análise estática, e que ela esteja disponível para ser utilizada no processo de análise dos projetos monitorados, termina de compor a característica extensível da ferramenta. A definição de uma interface que um *fetcher* tem de seguir, é um contrato estabelecido entre o núcleo do kiskadee e seus *fetchers*, e isso estabelece o monitoramento contínuo, essencial para que tenhamos uma análise da qualidade ao longo do tempo.

Os resultados apresentados demonstram que foi possível construir um sistema que responda a nossa questão-problema, tanto do ponto de vista de monitoramento quanto do ponto de vista de análise. Esse caráter extensível vem justamente da possibilidade de adicionar novos *fetchers* e novas ferramentas de análise estática, sem a necessidade de alterar diferentes partes do código da ferramenta. A versão 0.2.2 do kiskadee já possibilita acompanhar a qualidade de alguns projetos por meio das análises realizadas, e conforme novas ferramentas de análise estática forem adicionadas, a quantidade de falsos negativos que poderiam ocorrer utilizando apenas uma ferramenta, irão diminuir consideravelmente.

Como foi dito anteriormente, ferramentas de análise estática possuem seus problemas, como falsos positivos e negativos, mas também podem ser utilizadas de maneira efetiva para encontrar problemas sérios no código fonte. Realizar análises contínuas a cada nova versão lançada, permite que o desenvolvedor crie uma linha do tempo da qualidade do projeto, e analise se existe um esforço, por parte do time de desenvolvimento, em diminuir o número de fraquezas ou se a cada nova versão o número de fraquezas está aumentando. Viabilizar esse tipo de conclusão é o objetivo do projeto kiskadee, e esperamos que um número maior de desenvolvedores passem a considerar o uso de ferramentas como

a desenvolvida neste trabalho, que facilite a medição do produto, e que seja transparente ao desenvolvimento.

Além da participação no GSoC, a proposta de ferramenta que levou a criação do kiskadee também gerou uma publicação de um resumo expandindo para apresentação de pôster e demonstração da ferramenta no *13th International Conference on Open Source Systems*[1], a principal conferência acadêmica que trabalha o ecossistema de software livre.

## 6.1 Trabalhos Futuros

A arquitetura do kiskadee ainda precisa ser evoluída, e no contexto do GSoC o desenvolvimento irá continuar. O primeiro passo será desenvolver uma API para o kiskadee, de modo a permitir que outras aplicações possam consumir as análises feitas. A princípio não há uma intenção em desenvolver uma interface gráfica, devido a existência de um projeto chamado taskotron[2]. Esse projeto permite executar tarefas arbitrárias na infraestrutura do projeto Fedora. A ideia é que com o desenvolvimento da API do kiskadee, o taskotron possa requisitar as análises feitas, e apresentá-las de maneira amigável para o usuário. Caso essa integração não seja possível, uma interface gráfica será desenvolvida, mas que também irá consumir a API que iremos desenvolver. Um segundo passo no desenvolvimento será tornar a aplicação mais robusta do ponto de vista de tarefas que precisam ser executadas de maneira assíncrona, como os *fetchers* e os analisadores. Apesar do desenvolvimento do kiskadee está ocorrendo no contexto do GSoC, sob a supervisão do projeto Fedora, não existe um acoplamento do kiskadee à distribuição, de modo que qualquer um pode executar uma instância própria do projeto, na infraestrutura que lhe convir, desde que as dependências utilizadas para o monitoramento e análise, estejam disponibilizadas na infraestrutura em que o kiskadee irá ser utilizado. Todas essas atividades serão desenvolvidas até o final do GSoC, então em breve poderemos utilizar uma versão do kiskadee mais robusta e útil aos desenvolvedores interessados em análise estática de código fonte.

[1] http://oss2017.lifia.info.unlp.edu.ar/
[2] https://taskotron.fedoraproject.org/

# Apêndices

# APÊNDICE A – Revisão da Arquitetura do Debile

O Debile foi um projeto criado para atender uma demanda específica do Debian, sendo descrito como um sistema de *build*. É pequeno, minimo e implementado utilizando python. Isso permite que qualquer um use o Debile como uma plataforma de build de pacotes com extensão *.deb*, ou executando ferramentas customizadas tanto no código fonte quanto nos pacotes propriamente ditos. O repositório oficial do Debile pode ser encontrado no link <https://github.com/opencollab/debile.>

O Debian possui outras ferramentas específicas para construir pacotes binarios, como o Sbuild[1] e o Pbuilder[2], ferramentas mais apropriadas caso o objetivo seja construir pacotes a partir do código fonte. O grande atrativo em utilizar o Debile é a sua arquitetura distríbuida, e a facilidade de estendê-lo (por meio de extitfetchers) para outras situações, como por exemplo, para realizar análise estática nos pacotes fonte disponíveis nos repositórios do Debian.

### A.0.1 Arquitetura

O debile utiliza uma abordagem *master-slave*, bastante presente no processamento clusterizado. Nessa abordagem se assume que um trabalho $W$, de tamanho $m$, pode ser dividido em um conjunto $p$ de tarefas independentes $work_1, , , work_p$, de tamanhos arbitrários $m_1, , , m_p$, $\sum_{n=1}^{p} m_i = m$, que podem ser processados em paralelo pelos *slaves* $1, , , p$ (ALMEIDA et al., 2013). Após um slave processar a tarefa que lhe foi atribuída, ele retorna a resposta para o componente master, que sabe como reagrupar os resultados parciais obtidos pelos slaves.

O Debile segue uma política de controle em que cada slave deve ser registrado no banco de dados do master, de modo a termos um grupo fechado. Esse registro é feito incluindo o ip de cada slave na tabela *builders* do banco de dados. Com o grupo formado, o master disponibiliza *jobs* (pacotes novos ainda não incluídos no banco de dados), para serem processados pelos slaves. Um job será processado apenas por slaves devidamente configurados no banco de dados, e que possuam os *runners* ( extitfetchers) exigidos pelo grupo a qual um pacote pertence. O conceito de grupos no Debile é a maneira que foi estabelecida para definir quais runners deverão ser executados em determinados pacotes, sem a necessidade de ter que informar isso durante o processo de envio do pacote que será

[1] https://wiki.debian.org/sbuild
[2] https://pbuilder.alioth.debian.org/

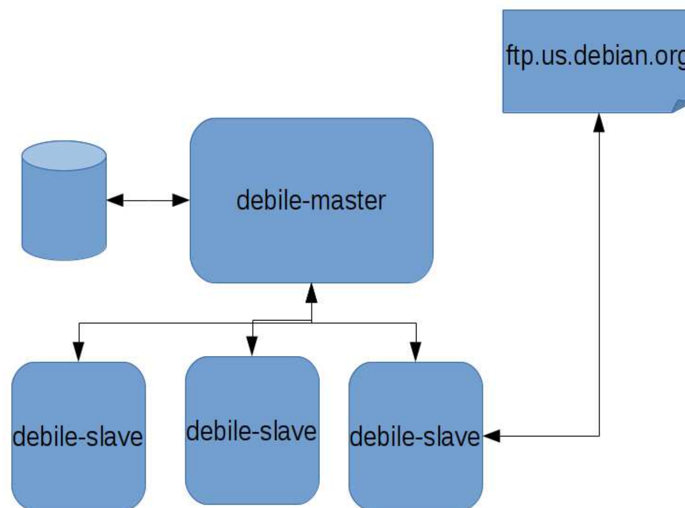



Figura 18 – Estrutura do Debile

analisado.

A tarefa de *upload* é responsável por trazer um pacote de um repositório Debian, para dentro da infraestrutura do Debile. Esse processo é feito por meio de duas ferramentas chamadas *dget* e *dput*. O dget é a ferramenta responsável por, a partir de um arquivo .dsc, fazer o download do código fonte do pacote. Após o download, o Debile assina o pacote com uma chave criada especificamente para isso, e com o dput envia o pacote para o componente master.

#### A.0.1.1 Runners

Os *runners* do Debile são utilizados pelos slaves para executar análises estáticas no código fonte dos pacotes. Quando um pacote qualquer se torna um job no master, os runners definidos para cada slave serão executados automaticamente. Quais runners serão executados, é uma informação definida em um arquivo de configuração lido pelo slave no momento em que ele é iniciado. O Debile não se limita apenas aos runners que se encontram no repositório oficial do projeto. Sua arquitetura permite que qualquer um implemente um novo runner, bastando apenas gerar um relatório no formato Firehose.

# APÊNDICE B – Catálogo de fraquezas de software

Existe uma padronização para que fraquezas e vulnerabilidades possam ser catalogadas e utilizadas em diferentes contextos. Essa padronização é feita por uma organização sem fins lucrativos que trabalha para o interesse público, chamada *Mitre*[1]. O Mitre é responsável por manter o projeto CWE (Common Weakness Enumeration), um dicionário de fraquezas comuns em software que podem ocorrer na arquitetura, projeto, código ou implementação, podendo levar a vulnerabilidades de segurança exploráveis. Todas as fraquezas presentes no projeto CWE possuem uma descrição padrão, incluindo um exemplo demonstrativo de como reproduzir tal fraqueza. Cada fraqueza catalogada possui um identificador que facilita a busca no dicionário. Este mesmo identificador pode ser utilizado por ferramentas de análise estática, estabelecendo uma linguagem comum entre o relatório da ferramenta e o que a comunidade de segurança registra no projeto.

No Exemplo B.1, é possível observar o identificador de determinada CWE, relacionada a uma ocorrência encontrada pela ferramenta de análise estática Cppcheck.

Listagem B.1 – Exemplo de erro coletado pelo Cppcheck

```
<error id="coutCerrMisusage" severity="error"
msg="Invalid usage of output stream: '<< std::cout'."
verbose="Invalid usage of output stream: '<< std::cout'." cwe="398">
       <location file="sentineal_search.cpp" line="27"/>
</error>
```

Ao buscar no dicionário pelo identificador 398, encontramos uma fraqueza com a seguinte descrição: *Indicação de código de baixa qualidade*. Havíamos comentado que não necessariamente uma fraqueza pode levar a uma vulnerabilidade, a fraqueza 398 é um exemplo prático disso. Fraquezas de software estão relacionadas a diversos aspectos do desenvolvimento de um produto, e vulnerabilidades de segurança estão inclusas nestes aspectos. É importante frisar que, a relação entre fraquezas e vulnerabilidades não necessariamente é direta. Uma vulnerabilidade pode ser composta por diferentes fraquezas de software, e uma fraqueza pode estar relacionada com outras fraquezas. Na taxonomia utilizada no projeto CWE, uma fraqueza pode ser *filha* ou *pai* de outras fraquezas. Utilizando a CWE 398 como exemplo, ela é filha da CWE 710 (Violação de padrões de código), e pai da CWE 399 (Erro de gerenciamento de recursos).

[1] https://www.mitre.org/

# APÊNDICE C – Arquivo de configuração do Kiskadee

```
[DEFAULT]
log_file = stdout

[db]
driver = postgresql
username = kiskadee
password = kiskadee
hostname = localhost
port = 5432
dbname = kiskadee

[debian_fetcher]
target = http://ftp.us.debian.org/debian
description = A fetcher to monitor Debian Repositories
release = sid
meta = Sources.gz
schedule = 1.0
analyzers = cppcheck
active = no

[juliet_fetcher]
target = juliet
description = SAMATE Juliet test suite
analyzers = cppcheck
active = no

[anitya_fetcher]
target = release-monitoring.org
description = A fetcher to monitor Anitya packages
analyzers = cppcheck
active = yes
zmq_port = 5556
zmq_topic = anitya

[example_fetcher]
target = example
description = SAMATE Juliet test suite
analyzers = cppcheck flawfinder
active = yes
```

```
[analyzers]
cppcheck = 1.0.0
flawfinder = 1.0.0
```

# APÊNDICE D – Trecho do arquivo Sources.gz

```
Package: curl
Binary: curl, libcurl3, libcurl3-gnutls, libcurl3-nss, libcurl4-openssl-
   dev, libcurl4-gnutls-dev, libcurl4-nss-dev, libcurl3-dbg, libcurl4-
   doc
Version: 7.52.1-3
Maintainer: Alessandro Ghedini <ghedo@debian.org>
Uploaders: Ian Jackson <ijackson@chiark.greenend.org.uk>
Build-Depends: debhelper (>= 9.20141010~), autoconf, automake, ca-
   certificates, dpkg-dev (>= 1.17.14~), groff-base, libgnutls28-dev,
   libidn2-0-dev, libkrb5-dev, libldap2-dev, libnghttp2-dev, libnss3-dev
   , libpsl-dev, librtmp-dev (>= 2.4+20131018.git79459a2-3~), libssh2-1-
   dev, libssl1.0-dev | libssl-dev (<< 1.1), libtool, openssh-server <!
   nocheck>, python:native, quilt, stunnel4 <!nocheck>, zlib1g-dev
Build-Conflicts: autoconf2.13, automake1.4
Architecture: any all
Standards-Version: 3.9.8
Format: 3.0 (quilt)
Files:
 a563149a0a0abf6815e1447f3879c7c5 2765 curl\_7.52.1-3.dsc
 4e1ef056e117b4d25f4ec42ac609c0d4 3504621 curl\_7.52.1.orig.tar.gz
 0483c1a94022eb89b0d504917d519c57 28912 curl\_7.52.1-3.debian.tar.xz
Vcs-Browser: https://anonscm.debian.org/gitweb/?p=collab-maint/curl.git
Vcs-Git: https://anonscm.debian.org/git/collab-maint/curl.git
Checksums-Sha256:
 80c2cf6870d658d5b8bc4b70c6d33b2e2c396d64b3cb2ea927d2f53fa1ac5e72 2765
    curl\_7.52.1-3.dsc
 a8984e8b20880b621f61a62d95ff3c0763a3152093a9f9ce4287cfd614add6ae
    3504621 curl\_7.52.1.orig.tar.gz
 e5a04a18e7728f3898da50845537123d3a500da7f959119eb36f1f73daee8cf7 28912
    curl\_7.52.1-3.debian.tar.xz
Homepage: http://curl.haxx.se
Package-List:
 curl deb web optional arch=any
 libcurl3 deb libs optional arch=any
 libcurl3-dbg deb debug extra arch=any
 libcurl3-gnutls deb libs optional arch=any
 libcurl3-nss deb libs optional arch=any
 libcurl4-doc deb doc optional arch=all
 libcurl4-gnutls-dev deb libdevel optional arch=any
 libcurl4-nss-dev deb libdevel optional arch=any
```

```
 libcurl4-openssl-dev deb libdevel optional arch=any
Directory: pool/main/c/curl
Priority: source
Section: libs
```

# Referências

ALMEIDA, F. et al. Modeling energy consumption for master–slave applications. 2013. Citado na página 55.

BLACK, P. E. Static analyzers in software engineering. *The Journal of Defense Software Engineering*, v. 22, n. 3, p. 16–17, 2009. Citado na página 22.

CHAHAR, C. et al. Code analysis for software and system security using open source tools. 2012. Citado 4 vezes nas páginas 11, 22, 23 e 24.

EMANUELSSON, P. A comparative study of industrial static analysis tools. 2008. Citado na página 21.

FEHNKER, A. et al. Fade to grey: Tuning static program analysis. 2010. Citado na página 22.

FENTON, N. et al. *Software metrics: a rigorous and practical approach*. Boston, USA, 1998. 54 p. Citado na página 21.

HECKMAN, S. et al. A systematic literature review of actionable alert identification techniques for automated static code analysis. 2011. Citado 2 vezes nas páginas 22 e 23.

IEEE. Ieee standard glossary of software engineering terminology. 1990. Citado na página 21.

KHAZAEI, A. et al. An automatic method for cvss score prediction using vulnerabilities description. 2016. Citado na página 17.

LANDI, W. Undecidability of static analysis. 1992. Citado na página 22.

OKUN, V. et al. *Static Analysis Tool Exposition (SATE) 2008*. Gaithersburg, USA, 2008. 64 p. Citado na página 22.

RENATUS, S. et al. Improving prioritization of software weaknesses using security models with avus. 2015. Citado na página 27.