Avaliação de Ferramentas de Análise Estática de Código para Detecção de Vulnerabilidades[1]

Emanuel Teixeira, João Antunes, Nuno Neves

Faculdade de Ciências da Universidade de Lisboa, Departamento de Informática
Campo Grande – Bloco C6 – Piso 3
1749-016 Lisboa, Portugal

emanuelteixeira@lasige.di.fc.ul.pt, {jantunes, nuno}@di.fc.ul.pt

Resumo

As ferramentas de análise estática facilitam a detecção de anomalias ou erros de codificação existentes numa aplicação. Estas ferramentas vêm ajudar a eliminar lapsos cometidos pelos programadores, podendo ter um impacto significativo no ciclo de desenvolvimento de um produto, permitindo poupar tempo e dinheiro.

Neste artigo é apresentado um teste que permite avaliar e comparar o desempenho de diversas ferramentas de análise estática, nomeadamente em relação ao número de falsos alarmes reportados e às vulnerabilidades que ficam por localizar. Os resultados obtidos com um conjunto de nove ferramentas demonstram que muitas delas estão especializadas para certas classes de vulnerabilidades, e que em média produzem um número significativo de falsos alertas. Daí ter-se concretizado uma nova ferramenta, designada por Mute, que utiliza um mecanismo de agregação de resultados produzidos por vários outros analisadores. Uma avaliação do Mute mostrou que ele era capaz de apresentar uma melhor eficácia e precisão na detecção para um conjunto alargado de vulnerabilidades.

1 Introdução

As falhas na programação de aplicações podem dever-se a diversas razões como interacções pouco claras entre módulos, complexidade do código a desenvolver, e falta de diligência dos programadores (por exemplo, o uso de funções de biblioteca que potenciam o aparecimento de vulnerabilidades [9]). Torna-se assim imprescindível a utilização de mecanismos de revisão eficaz do código fonte, que facilitem a descoberta e remoção dos erros.

Um dos primeiros passos para a eliminação de vulnerabilidades passou pela revisão manual de código fonte. Nesta técnica, utilizada há mais de duas décadas, um conjunto de auditores examinam pormenorizadamente o código da aplicação, de modo a encontrar possíveis falhas de segurança. Este processo acaba por ser muito moroso e caro, daí surgirem mecanismos de revisão por meio computacional, como a análise estática. Neste tipo de análise, uma ferramenta estuda o programa linha a linha à procura de erros, e produz um relatório que descreve os problemas que potencialmente podem existir. Em seguida, o programador tem apenas de verificar as partes do código para as quais aparecem avisos.

As ferramentas que concretizam este tipo de análise devem fazer parte do ciclo de desenvolvimento de um produto, pois têm características muito ricas e são capazes de eliminar muitos dos erros, diminuindo o tempo de testes e de correcção [2]. Elas podem ser empregues durante a escrita do código fonte, uma vez que não necessitam que a aplicação esteja concluída para poderem funcionar. Assim, por exemplo, sempre que um programador termina a programação de um módulo, pode testá-lo individualmente, evitando que certos erros passem para a próxima fase de desenvolvimento. Como seria espectável, a análise

[1] Este trabalho foi suportado parcialmente pela União Europeia através dos projectos *CRUTIAL* (IST-4-027513-STP), *RESIST* (IST-4-026764-NOE), e também pela FCT por meio do projecto *AJECT* (POSC/EIA/61643/2004) e do Laboratório de Sistemas Informáticos de Grande-Escala (LASIGE).

estática também pode ser usada na fase de teses, normalmente em conjunto com outros tipos de ferramentas, uma vez que nessa altura se tem acesso à aplicação completa.

Actualmente existem várias ferramentas de detecção de vulnerabilidades por análise estática disponíveis na *Internet*. No entanto, estas estão vocacionadas para encontrarem certos tipos específicos de vulnerabilidades e utilizam diversas técnicas para fazerem a detecção (por exemplo, análise sintáctica, provadores de teoremas, etc.). Daí ser extremamente difícil a comparação das capacidades efectivas de cada ferramenta, tornando impossível a escolha entre elas. A fim de avaliar as ferramentas existentes, desenvolveu-se uma versão preliminar de um teste padronizado (do inglês, *benchmark*) que permitisse aferir e comparar o desempenho das ferramentas. Optou-se por se focar o estudo em ferramentas de análise da linguagem C, porque esta é uma das linguagens mais usadas[1], especialmente ao nível da programação do sistema. Por outro lado, como o C permite uma grande flexibilidade, facilita a introdução de vulnerabilidades no código.

A definição do teste obrigou à realização de diversas tarefas. Nomeadamente, foi necessário proceder-se a uma investigação das vulnerabilidades que estão actualmente a ser exploradas pelos piratas informáticos. Como resultado desta investigação, foi definido um conjunto extensivo de classes de vulnerabilidades, que inclui problemas como a validação de parâmetros e a conversão explícita de tipos (cast). Outra tarefa consistiu no desenvolvimento de um programa vulnerável que teria de ser entregue às ferramentas para análise. Este programa contém muitos exemplos representativos das classes de vulnerabilidades, e para além disso, alguns pedaços de código que estão certos, mas que muitas vezes são incorrectamente indicados como erros. Este segundo grupo de linhas de código é de extrema importância para determinar se as ferramentas geram muitos falsos avisos (também chamados de *falsos positivos*). Por fim, foram definidas as métricas que possibilitam comparar o desempenho das ferramentas, nomeadamente a eficácia e a precisão, que são calculadas a partir dos relatórios produzidos.

O teste foi usado para avaliar nove ferramentas existentes. Observou-se que as ferramentas encontram-se normalmente especializadas para a detecção de certas classes específicas de vulnerabilidades, não havendo alguma capaz de localizar um grande número de classes. Logo, um indivíduo tem de recorrer a várias ferramentas se quer garantir a eliminação de diversas vulnerabilidades. Adicionalmente, notou-se que as ferramentas costumam produzir falsos alertas, tornando-se assim importante criar métodos que de certa forma possam reduzir o seu número (a investigação de um falso alerta acaba por consumir tempo que poderia estar a ser usado em alguma actividade útil).

Tendo em consideração estes resultados, desenhou-se uma nova ferramenta chamada Mute, que através da agregação de resultados de diversas ferramentas, gera um novo relatório. É de notar, que a escolha da forma como faz a agregação de resultados acaba por ter alguma complexidade, porque o objectivo seria que o Mute tivesse um menor número de falsos positivos, mantendo os verdadeiros positivos (i.e., as vulnerabilidades correctamente localizadas). A avaliação do Mute através do teste padronizado demonstrou que este objectivo era passível de ser atingido.

Em resumo, o artigo tem duas contribuições fundamentais. Ele apresenta um teste para a avaliação de ferramentas de análise estática de código fonte C, e descreve os resultados da sua aplicação a nove ferramentas bem conhecidas. Em acréscimo, propõe-se uma nova ferramenta que é construída a partir da agregação de resultados de outras ferramentas, mas que potencialmente detecta um maior número de vulnerabilidade, mantendo os falsos positivos dentro de valores aceitáveis.

[1] Classificação das linguagens em, `http://www.tiobe.com/tpci.htm`.

2 Teste de Avaliação

Como é possível constatar a partir da amostra apresentada na secção 4.2, existem neste momento várias ferramentas de análise estática de código fonte disponíveis na *Internet*. No entanto, em alguns casos, estas ferramentas resultam de projectos de investigação, normalmente têm um suporte limitado e a documentação é incompleta. Torna-se assim difícil a identificação das suas verdadeiras capacidades de detecção. Por outro lado, actualmente não se encontram disponíveis bons métodos que permitam avaliar e comparar as ferramentas quanto ao seu funcionamento, o que complica a escolha das ferramentas a usar.

Um teste permite avaliar e comparar sistemas ou componentes de acordo com determinadas características, de uma maneira simples e com resultados facilmente compreensíveis pelos seus utilizadores. Genericamente, as características a avaliar nas ferramentas de análise estática são o número e tipo de vulnerabilidades que são detectadas, e o número de falsos alertas reportados. A vantagem de se recorrer a um teste consiste em se identificar a aptidão de cada ferramenta através de um método reproduzível, e assim, por exemplo desmistificar ferramentas que garantem a localização de 100% de vulnerabilidades.

2.1 Propriedades

Um teste padronizado deve satisfazer um conjunto de propriedades para que seja útil [10]. Essas encontram-se resumidas em seguida:

Representatividade: O teste devolve resultados interessantes se representar os tipos de problemas que são observados na realidade. Caso contrário, os resultados não caracterizam uma utilização real das ferramentas de análise. No nosso caso, o teste padronizado deve-se basear num conjunto adequado de vulnerabilidades a detectar, e as medidas devem reflectir a capacidade efectiva na localização de erros que realmente existem (por oposição a falsos alertas).

Portabilidade: A portabilidade num teste significa que este deve permitir a comparação equitativa entre diferentes ferramentas, pertencentes ao mesmo domínio de aplicação, em ambientes de execução com características distintas. Através desta avaliação consegue-se verificar a aplicabilidade do teste padronizado em sistemas muito diferentes.

Reprodutibilidade: Entende-se por reprodutibilidade de um teste à capacidade de produzir resultados similares, quando executado mais do que uma vez no mesmo ambiente. Uma vez que as ferramentas de análise estática que iremos considerar estudam os códigos fonte de uma maneira determinística, onde só podem apresentar variações quanto ao tempo de execução, mantendo as mesmas capacidades de detecção.

É possível comprovar que um teste padronizado é reprodutível através de várias execuções no mesmo sistema, sendo que, os resultados deverão ser analisados e as potenciais variações identificadas. Se o ambiente de execução for diferente, então dependendo da ferramenta é possível que a sua execução não seja igual.

Escalabilidade: Um teste deve ser capaz de avaliar uma ferramenta com cargas (ou programas) de diferentes dimensões. A forma como a escala pode ser alterada/medida é através do aumento de linhas de código a processar, com ou sem novas vulnerabilidades inseridas.

Não-intrusividade: Esta propriedade exige que um teste padronizado não obrigue a alterações nas ferramentas para que ele possa ser executado. A propriedade de não-intrusividade é normalmente fácil de verificar, quando se aplica o teste sobre uma ferramenta e esta necessita de uma configuração específica para ser utilizada.

Simplicidade de Utilização: Um teste deve ser fácil de usar na prática, estando a sua potencial aceitação pela comunidade fortemente relacionada com esta propriedade. Para além disso, a execução de um teste deve ser pouco dispendiosa, completamente automática e não deve consumir demasiado tempo.

2.2 Classes de Vulnerabilidades

Ao longo deste trabalho foram estudados vários tipos de vulnerabilidades, com efeitos potencialmente diversos na segurança dos programas. Em baixo podem-se observar as diferentes categorias de vulnerabilidades que foram consideradas para inclusão no teste.

1. **Inicialização** (INIT) - Tipos de dados não inicializados;
2. **Conversões** (CAST) - Incorrecta conversão entre tipos de dados;
3. **Validação de Parâmetros** (PVAL) - Dados de parâmetros contêm irregularidades;
4. **Memória Sobrelotada** (BUFFER) - Escrever em zonas de memória ilegais (inclui os bem conhecidos *buffer overflows*);
5. **Formatação de Cadeias de Caracteres** (FORMAT) - Má formatação em cadeias de caracteres (inclui os erros de *format strings*);
6. **Acesso Fora dos Limites** (BOUND) - Acesso de leitura para além do limite estipulado numa estrutura de dados (exemplo, *array*);
7. **Exceder o Espaço de Representação de um Inteiro** (INTEGER) - Modificação do valor do inteiro para um valor incorrecto do previsto;
8. **Divisão por Zero** (ZERO) - Falha na validação do divisor;
9. **Ciclos Infinitos** (LOOP) - Ocorrências de ciclos infinitos, devido à má validação;
10. **Valor de Retorno** (RET) - Necessidade de validar o valor de retorno;
11. **Gestão de Memória** (MEM) - Má gestão e concretização dos recursos de memória;
12. **Acesso a Ficheiros** (FILA) - Incorrecto acesso a ficheiros;
13. **Redução de Privilégios** (LEAST) - Necessário diminuir os privilégios do utilizador;
14. **Números aleatórios** (RANDOM) - Geração de números aleatórios com limitações (útil para criação de chaves de cifra seguras).

A cada uma destas classes de vulnerabilidades foram associados vários exemplos práticos (pedaços de código) onde elas se materializam. A cada momento é possível adicionar (e/ou remover) novos excertos de código vulnerável, dado que, a lista actual se encontra em evolução. A versão actual da lista de vulnerabilidades foi construída através de material recolhido de diferentes fontes. Nomeadamente, foram estudados os programas teste que eram fornecidos com as ferramentas, e foram investigados vários sítios na *Web* que se dedicam a disponibilizar informação sobre a descoberta de novas vulnerabilidades.

Esta categorização das vulnerabilidades acaba por ser muito útil porque permite garantir que cada classe tem um número mínimo de exemplos representativos. Adicionalmente, é interessante porque permite rapidamente descobrir o grau de especialização de cada ferramenta, para a localização de tipos específicos de vulnerabilidades.

2.3 Medidas de Avaliação

O resultado da execução de uma ferramenta de análise é um relatório que indica as linhas do programa onde possivelmente podem existir vulnerabilidades. Alguns destes alertas podem estar errados, porque houve um engano da ferramenta, e são chamados de *falsos positivos*. Os restantes alertas correspondem a vulnerabilidades que realmente existem, e por isso são designados de *verdadeiros positivos*. Por outro lado, podem existir vulnerabilidades que não

são indicadas no relatório, ou seja que não são descobertas pela ferramenta, intitulados de *falsos negativos*. Uma ferramenta será tão mais útil quanto menor for o número de falsos negativos e falsos positivos. Baseando-se nestas ideias, usaram-se as seguintes métricas para avaliar o desempenho das ferramentas (inspiradas no trabalho de Dominique [1]):

- *Eficácia*: é a percentagem do número total de vulnerabilidades detectadas por uma ferramenta. O número total de vulnerabilidades correctamente reportadas pela ferramenta corresponde aos verdadeiros positivos (m_d). O número total de vulnerabilidades existentes é a soma entre as vulnerabilidades detectadas e as não detectadas (m_{nd}). A eficácia é calculada através da fórmula:

$$e = \frac{m_d}{m_d + m_{nd}}. \tag{1}$$

- *Precisão*: é a percentagem do número total de vulnerabilidades detectadas pela ferramenta que corresponde aos verdadeiros positivos. O número total de alertas é a soma do número de verdadeiros positivos (a_{vp}) mais o número de falsos positivos (a_{fp}):

$$p = \frac{a_{vp}}{a_{vp} + a_{fp}}. \tag{2}$$

As duas métricas dão-nos informação sobre a confiança dos alertas gerados pelas ferramentas de análise estática. Utilizaram-se estas duas métricas porque englobam os dados mais relevantes para o estudo, e facilitam a comparação das ferramentas através de dois números.

2.4 Versão Actual do Teste

A versão actual do teste garante praticamente todas as propriedades indicadas anteriormente. O teste é representativo porque contém diversos tipos de problemas que são observados na realidade, isto é, contém um conjunto alargado de vulnerabilidades. Quanto à portabilidade, o teste só foi testado para o ambiente Linux, no entanto, não se antevê qualquer dificuldade quanto à sua execução em outro ambiente, desde que, se mantenham os requisitos que inicialmente foram dispostos, ou seja, um compilador semelhante ao gcc 3.4.2. A reprodutibilidade é cumprida porque este não sofre variações durante a sua execução, reproduzindo invariavelmente as mesmas vulnerabilidades quando executado mais que uma vez. Relativamente à escalabilidade, o teste não garante esta propriedade porque o programa vulnerável tem um tamanho fixo. O teste não necessita de nenhuma alteração/modificação para ser executado pelas ferramentas, daí a não-intrusividade. Por fim, a simplicidade de utilização é elevada porque o teste é fácil de trabalhar, ou seja, de acrescentar novos pedaços de código vulnerável, e cada vulnerabilidade está identificada para o seu tipo, podendo assim qualquer pessoa produzir resultados a partir deste e mesmo aumentando a qualidade do teste com mais vulnerabilidades. Porém, à medida que o teste for crescendo com mais funções vulneráveis, convém dividir cada categoria de vulnerabilidades por um ficheiro diferente, para um melhor aproveitamento.

Neste momento estão concretizados exemplos de código vulnerável para cada uma das classes, sendo o número total de erros que foi introduzido no programa a analisar de 264[1]. Junto com as linhas vulneráveis, foram incluídos também alguns pedaços de código correcto que tradicionalmente causam problemas de interpretação às ferramentas, i.e., que são muitas

[1] A versão actual do programa que contém as vulnerabilidades está disponível em, `http://aject.di.fc.ul.pt/code/static_analysis.zip`.

vezes reportados erradamente como vulnerabilidades. A recolha destes excertos de código fez-se através de uma análise apurada dos programas de teste que são fornecidos com as distribuições das ferramentas, e que por isso foram escolhidos pelos seus autores para avaliar as capacidades das ferramentas. Desta maneira, garante-se que o teste não só averigua a eficácia na localização de vulnerabilidades, mas também o nível de precisão com que essa detecção é efectuada.

3 A Ferramenta Mute

As ferramentas de análise estática têm como objectivo localizar vulnerabilidades existentes nos programas, e que possam pôr em causa a segurança. A grande vantagem de se usar estas ferramentas é rever o código desenvolvido muito antes de este ser entregue para testes.

Do nosso estudo sobre diversas ferramentas disponíveis na *Internet* (ver Secção 4) foi possível observar que existem ferramentas vocacionadas para encontrar tipos muito específicos de vulnerabilidades, e que empregam distintas técnicas de análise. É de notar, no entanto, que não há uma ferramenta que analise todas as categorias de vulnerabilidades. Adicionalmente, verificou-se que as ferramentas reportam um número não desprezável de falsos positivos, algo que se gostaria de evitar.

Para se tentar resolver, ainda que parcialmente, estas questões, desenvolveu-se uma ferramenta denominada por *Mute* que usa várias ferramentas com distintos aspectos de análise e classes de vulnerabilidades. Genericamente, o Mute recolhe os resultados de análise de cada ferramenta, também designados por relatórios, e formata-os num modelo pré-definido. A partir daqui, agrega os resultados de acordo com um determinado algoritmo, e depois decide-se que vulnerabilidades devem ser reportadas ao utilizador.

É de notar que o problema da agregação de resultados não é trivial uma vez que se pretende maximizar quer a precisão como a eficácia. Assim, algoritmos simplistas como a "junção de todas as vulnerabilidades indicadas pelas ferramentas", acabam por dar resultados pouco satisfatórios. Neste exemplo específico, obtinha-se o maior valor possível para a eficácia, mas ter-se-ia uma baixa precisão.

3.1 Arquitectura do Mute

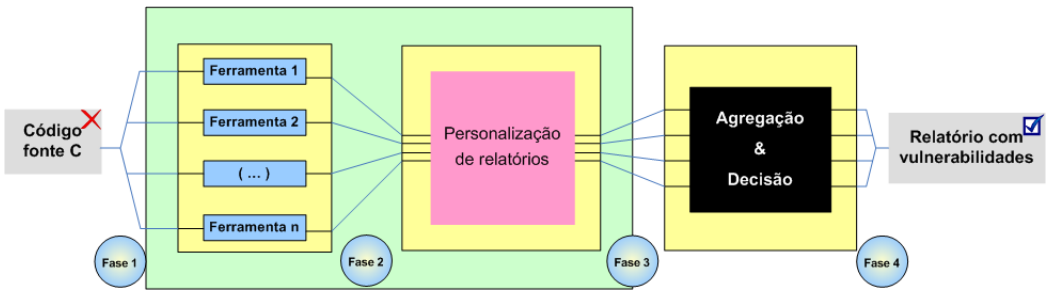

Figura 1: Arquitectura da ferramenta Mute.

Na Figura 1 tem-se uma representação da arquitectura da ferramenta desenvolvida. Como se pode observar, o Mute encontra-se organizado em dois módulos independentes. O primeiro módulo divide-se em duas componentes, em que a primeira trata da recepção do código fonte C, possivelmente vulnerável, e entrega-o às ferramentas para análise (Fase 1). Depois são extraídos os relatórios gerados por cada ferramenta, alterando a sua formatação para um modelo mais genérico (Fase 2). O segundo módulo, que é totalmente independente do primeiro, recebe os vários relatórios (Fase 3) e produz um relatório temporário onde se

encontram todas as ocorrências dos avisos. Por fim, é elaborado o relatório final através da aplicação do algoritmo de agregação (fase 4). Esta arquitectura acaba por ser modular e portável, uma vez que módulos têm uma interface bem definida, e porque é independente da linguagem de programação analisada.

3.2 Ferramentas Usadas e Tratamento dos Relatórios

Nesta subsecção descrevem-se as duas componentes do primeiro módulo da Figura 1. Este módulo foi concretizado por meio da linguagem *Shell Script*, por ser uma linguagem fácil de trabalhar, uma vez que é possível configurar e executar de forma simples as aplicações na linha de comandos.

O primeiro componente recebe a localização do(s) ficheiro(s) de código a tratar pelas ferramentas e procede à sua análise. A análise é feita de forma sequencial, uma análise seguida da próxima, e assim sucessivamente até não haver mais ferramentas e/ou ficheiros de código. Cada ferramenta deixa o seu relatório com os avisos das possíveis vulnerabilidades numa directoria específica.

A segunda componente pega nos relatórios e formata-os num modelo específico, através da execução de comandos UNIX. Após a formatação, cada relatório é copiado para uma outra directoria para o segundo módulo analisar.

```
ficheiro.c:linha:ferramenta:vulnerabilidade:descrição
descrição
(...)
ficheiro.c:linha:ferramenta:vulnerabilidade:descrição
(...)
```

Figura 2: Modelo de formatação do relatório pelo Mute.

Um ficheiro de relatório deverá seguir o modelo apresentado na Figura 2. A informação fundamental a manter é: o nome do ficheiro analisado; a linha onde se encontrou a vulnerabilidade; o nome da ferramenta de análise; o tipo de vulnerabilidade que a ferramenta detectou, que corresponde a uma das classes de vulnerabilidades (ver Secções 2.2); por fim, pode-se ter uma descrição sobre o problema encontrado na mesma linha ou na(s) linha(s) seguinte(s).

3.3 Agregação e Processamento de Resultados

Nesta subsecção descreve-se o segundo módulo apresentado na Figura 1. O módulo foi desenvolvido em Java, uma vez que esta linguagem possuí um conjunto de API's que possibilitam uma fácil e rápida concretização das tarefas a executar (e.g., processamento de cadeias de caracteres, árvores, etc).

A ferramenta carrega em memória os relatórios, e durante a leitura, verifica se a linha corresponde ao formato especificado. No caso afirmativo, adiciona-a a uma tabela de dispersão, tendo o cuidado de detectar os casos em que a vulnerabilidade já tinha aparecido num relatório anterior. Nesta situação, o Mute simplesmente acrescenta mais dados à entrada existente. Depois de ler todos os relatórios, tem-se uma tabela com toda informação, que é então usada para se produzir o relatório final. Para o construir basta ir a todas as ocorrências da tabela e avaliar-se se existe confiança no aviso correspondente. De acordo com essa avaliação, é então decidido se se deve ou não adicionar uma entrada no relatório final, eliminando-se assim os possíveis falsos alertas.

No algoritmo de decisão existente, a avaliação das entradas na tabela baseia-se num valor de confiança ou certeza sobre os resultados produzidos por cada ferramenta (ver

exemplos práticos na Secção 4.4). Esse *nível de confiança* é calculado através do número de falsos positivos *versus* verdadeiros positivos que são em média observados com essa ferramenta, e tem um valor entre 0.0 e 1.0. Assim, quando se afirma uma confiança de 80% numa ferramenta é porque esta consegue localizar muitas vulnerabilidades com um grau de certeza muito grande, ou seja, com poucos falsos positivos.

Uma entrada na tabela pode corresponder a uma vulnerabilidade detectada por uma ou várias ferramentas. O valor de confiança[1] a atribuir à entrada, e logo ao aviso correspondente, é calculado usando uma média ou uma soma (dependo do algoritmo) dos níveis de confiança de cada ferramenta. Se o valor resultante for igual ou superior a uma constante pré-definida, por exemplo 0.5, então o aviso é considerado válido e é adicionada informação sobre esta vulnerabilidade no relatório a devolver ao utilizador. Caso contrário, a entrada é rejeitada.

O Mute actualmente avalia as ferramentas com base em dois tipos de confiança, uma genérica relacionada com a ferramenta, ou seja, com a precisão que esta tem em detectar vulnerabilidades; e outra relativa a cada classe de vulnerabilidade que a ferramenta detecta, isto porque há ferramentas especializadas para certas classes de vulnerabilidades, e como tal tem mais sucesso nessa classe do que no conjunto de diferentes vulnerabilidades que pode detectar. Estes níveis são fornecidos ao Mute para o configurar, no entanto, o seu valor pode variar no futuro à medida que se ganha mais experiência com uma determinada ferramenta.

4 Resultados Experimentais

Nesta secção procede-se a duas avaliações utilizando o teste, uma sobre as ferramentas estudadas e outra sobre o desempenho do Mute. Um dos objectivos desta avaliação é a recolha de dados que ajudem a mostrar que uma aproximação semelhante à do Mute potencialmente consegue encontrar mais vulnerabilidades que cada ferramenta individualmente, com um número de falsos positivos reduzido (existe, no entanto, sempre a possibilidade de algumas vulnerabilidades deixam de ser localizadas).

4.1 Ambiente de Execução

A avaliação foi conduzida num computador Fujitsu Siemens, com processador Intel Pentium IV a 2.80GHz de 32 bits e 512MB de memória. O sistema operativo era o Linux, com a distribuição Fedora Core 3.0 e o kernel 2.6. O compilador utilizado no ambiente de testes foi o gcc 3.4.2.

4.2 Ferramentas Testadas

Existem várias ferramentas de análise estática de código fonte, quer comerciais como associadas a projectos de investigação [11]. Para este trabalho só foram utilizadas e testadas ferramentas disponíveis na *Internet*, pois estas são gratuitas. Ferramentas distintas conseguem encontrar diferentes vulnerabilidades, isto devido à técnica de análise utilizada e também aos tipos de vulnerabilidades para os quais as ferramentas foram desenvolvidas. No entanto, existem ferramentas com técnicas diferentes, mas dirigidas para o mesmo tipo de vulnerabilidades, que encontram vulnerabilidades reais diversas, para além das em comum.

[1] Que é baseado na precisão que uma ferramenta consegue detectar vulnerabilidades. Neste valor é tido em conta os verdadeiros positivos *vs.* falsos positivos reportados.

4.2.1 Crystal

A ferramenta Crystal (versão 1.0)[1] é a mais recente de todas as que foram testadas e só foi disponibilizada no fim do ano de 2006. Foi desenvolvida na Universidade de Cornell [6].

O Crystal é uma plataforma de suporte com diversas extensões, no entanto, até ao momento só está concretizada a detecção de vulnerabilidades relacionadas com a gestão de memória (encontram-se em desenvolvimento duas novas extensões). Esta ferramenta trata da vulnerabilidade de gestão de memória, por exemplo, reservar espaço na memória n vezes sem libertar o que foi obtido anteriormente.

A ferramenta foi programada na linguagem Java para detectar vulnerabilidades em código fonte C e utiliza diferentes técnicas de análise de código, como árvores abstractas de sintaxe, controlo do fluxo por grafos e a análise de fluxo de dados [7]. Além destas técnicas, consegue ainda obter invariantes do código fonte a partir da análise dentro e entre procedimentos (potencialmente localizados em ficheiros diversos).

4.2.2 Deputy

O Deputy (versão 1.0)[2] é baseado em anotações, embora, consiga detectar alguns problemas, como memória sobrelotada (do inglês, *buffer overflow* ou *underflow*), sem necessitar das anotações [4]. Foi desenvolvida por Jeremy Condit et al. na Universidade de Berkeley. A ferramenta usa vinte e uma anotações[3] a adicionar no código fonte C. Utiliza uma linguagem intermédia de alto nível, o CIL[4], para analisar e transformar código fonte.

O Deputy tem um número razoável de anotações para variáveis, apontadores e estruturas de dados (*struct* ou *union*). Para os autores, o uso de anotações é uma mais-valia da ferramenta porque são poucas e fáceis de usar, o que facilita o trabalho dos programadores.

A vulnerabilidade que a ferramenta detecta mais é a memória sobrelotada, embora também detecte problemas de formatação de cadeias de caracteres (do inglês, *format string*) e erros comuns de programação.

4.2.3 Flawfinder

O Flawfinder (versão 1.26)[5] é uma ferramenta de análise de padrões, desenvolvida por David Wheeler na linguagem de programação Python, sendo muito semelhante às ferramentas ITS4 e RATS. A ferramenta suporta código nas linguagens C e C++.

Uma vez que esta ferramenta analisa o código fonte em busca de padrões, ela contém uma base de dados com funções potencialmente vulneráveis. Estas funções correspondem a vulnerabilidades do tipo memória sobrelotada, formatação de cadeias de caracteres, controlo de acesso (do inglês, *access control*), condições de disputa (do inglês, *race condition*) e números aleatórios.

4.2.4 ITS4

O ITS4 (versão 1.1.1)[6] foi uma das primeiras ferramentas propostas para a detecção de vulnerabilidades através da análise de padrões [8]. A ferramenta foi criada para substituir o comando *grep* do Unix, tornando a busca mais fácil e rápida de vulnerabilidades. O ITS4 foi desenvolvido por um conjunto de investigadores da Reliable Software Technologies (Cigital) e concretizado em C++. A ferramenta analisa programas em C e C++, e utiliza uma base de dados com uma lista de funções vulneráveis.

[1] Encontra-se em, `http://www.cs.cornell.edu/projects/crystal/`.
[2] Tem uma página *web* em, `http://deputy.cs.berkeley.edu/`.
[3] As anotações são tipos qualificativos, muito semelhantes ao *const* em C.
[4] Mais informação consulte, `http://hal.cs.berkeley.edu/cil/`.
[5] Mais informação em, `http://www.dwheeler.com/flawfinder/`.
[6] Sobre a ferramenta ITS4 em, `http://www.cigital.com/services/its4`.

A única técnica de análise é a busca de padrões e não utiliza outros conceitos como uma árvore abstracta de sintaxe, isto porque, para os autores, era importante tornar a ferramenta rápida, simples e útil no desenvolvimento de uma aplicação [8]. O ITS4 pesquisa por diversas vulnerabilidades, tais como, memória sobrelotada, formatação de cadeias de caracteres, controlo de acesso, condições de disputa e números aleatórios.

4.2.5 MOPS

A ferramenta MOPS (versão 0.9.2)[1] foi desenvolvida por Hao Chen e David Wagner, e até ao momento ainda continua numa fase inicial de desenvolvimento [3].

O MOPS foi construído para a linguagem de programação C, e utiliza duas técnicas de análise sofisticadas, a análise por fluxo de dados e o controlo de fluxo por grafos. Detecta as vulnerabilidades a partir de um conjunto de propriedades, onde cada propriedade corresponde a um tipo particular de vulnerabilidade (exemplo, a validação do *strcpy()* é diferente do *strncpy()*) e são estas que verificam se há alguma sequência de operações que mostram um comportamento inseguro. Até ao momento, estão disponíveis vinte e duas propriedades.

4.2.6 PScan

PScan (versão 1.2)[2] é uma ferramenta de análise estática de código fonte para detecção de vulnerabilidades de formatação de cadeias de caracteres e memória sobrelotada (ou seja, família *printf()* e *scanf()*). É constituída por uma base de dados com uma lista de funções possivelmente vulneráveis para a linguagem de programação C.

A técnica de análise utilizada pelo PScan é a análise lexical, que procura por funções vulneráveis e examina os seus argumentos. A desvantagem desta ferramenta é o número limitado de vulnerabilidade que consegue detectar, o que nem sempre é útil quando se pretende analisar código fonte com diferentes tipos de vulnerabilidades. No entanto, é uma ferramenta rápida e com um bom grau de precisão.

4.2.7 RATS

RATS (versão 2.1)[3] é uma ferramenta com semelhanças ao Flawfinder e ITS4, sendo considerada um sucessor do ITS4. A ferramenta foi desenvolvida na linguagem de programação C e utiliza a análise de padrões. Detecta vulnerabilidades de memória sobrelotada, formatação de cadeias de caracteres, controlo de acesso, condições de disputa e números aleatórios. A análise do código fonte pode ser feita sobre as seguintes linguagens de programação: C, C++, Perl, PHP e Python.

Esta ferramenta também requer uma base de dados com funções vulneráveis, no entanto, comparativamente a outras, possui uma lista maior de funções vulneráveis.

4.2.8 Sparse

O Sparse (versão 0.2)[4] é um analisador semântico para a linguagem de programação C na norma ANSI. É muito semelhante aos analisadores semânticos utilizados nos compiladores. O Sparse reporta problemas existentes no código fonte que vão contra a norma ANSI, detectando também problemas entre tipos de dados e acesso fora do limite de uma lista.

[1] O *site* oficial em, `http://www.cs.ucdavis.edu/~hchen/mops/`.
[2] Encontra-se em, `http://www.striker.ottawa.on.ca/~aland/pscan/`.
[3] A ferramenta encontra-se em, `http://www.fortifysoftware.com/security-resources/rats.jsp`.
[4] Para obter a ferramenta e mais dados consulte, `http://www.kernel.org/pub/software/devel/sparse/`.

O Sparse foi desenvolvido por Linus Torvalds a partir de 2003, e tinha como objectivo inicial detectar falhas com apontadores. Neste momento, a ferramenta continua em desenvolvimento, agora por Josh Triplett, e encontra-se na versão 0.3, que foi disponibilizada em Maio de 2007. O seu desenvolvimento está muito condicionado porque se pretende que jovens estudantes possam aplicar as suas ideias neste projecto.

4.2.9 UNO

O UNO (versão 2.11)[1] é um analisador estático para código C na norma ANSI [5]. Começou a ser desenvolvido em 2001, por Gerard Holzmann; a última versão 2.12 foi lançada em Agosto de 2007. O UNO usa um conjunto de propriedades pré-definidas e sobre elas analisa a aplicação. Dá oportunidade aos utilizadores de definirem as suas propriedades e de estas serem utilizadas durante a análise, funcionando de maneira semelhante ao MOPS. Permite também escolher a forma de análise, global ou local, isto é, analisa localmente ficheiro a ficheiro ou todos os ficheiros.

A ferramenta consegue detectar três tipos de vulnerabilidades: a não inicialização de uma variável, referência nula de um apontador e acesso fora dos limites de uma lista. Nas três vulnerabilidades que consegue detectar fornece bons resultados, com um elevado grau de precisão.

4.3 Avaliação Individual das Ferramentas

Nesta avaliação pretende-se determinar as capacidades das ferramentas em termos de localização de vulnerabilidades (verdadeiros positivos), as que não detectam (falsos negativos) e as que são detectadas erradamente (falsos positivos). Todas as ferramentas partem de uma situação idêntica, ou seja, todas vão examinar o mesmo ficheiro de código fonte C em busca de vulnerabilidades. Só assim se pode comparar as ferramentas, até mesmo aquelas que utilizam a mesma técnica de análise e/ou procuram as mesmas vulnerabilidades.

Todas as ferramentas produzem um relatório, no entanto, este não contém a informação relativa à veracidade dos alertas. Por isso, analisou-se cada relatório manualmente, linha a linha, comparando as vulnerabilidades previamente identificadas com as vulnerabilidades reportas pela ferramenta. Deste modo, foi reconhecida a existência ou não de avisos, uma vez que no teste já se tinham identificado todas as vulnerabilidades possíveis de ocorrer, tendo bastado apenas comparar e contabilizar as vulnerabilidades correctamente descobertas ou em falta.

Os resultados obtidos para cada ferramenta são apresentados na Tabela 1. Da análise dos resultados percebe-se que não há nenhuma ferramenta a detectar todos os tipos de vulnerabilidades, assim como, são pouquíssimas as ferramentas a detectarem a totalidade das vulnerabilidades de uma classe. Isto demonstra que ainda há muito espaço para melhoramentos na detecção estática de vulnerabilidades.

Uma ferramenta que sobressai é o Crystal, responsável por detectar vulnerabilidades de gestão de memória, a partir de uma análise de fluxo de dados e controlo do fluxo por grafos. Estas duas técnicas levam a que a ferramenta seja muito precisa, embora tenha um baixo valor de eficácia, porque está muito especializada para encontrar problemas de gestão de memória, deixando as outras vulnerabilidades por detectar. A ferramenta com piores resultados é o ITS4, devido à utilização da análise de padrão. O ITS4 tem um problema que acabou por o influenciar negativamente, que é o elevado número de falsos positivos na formatação da cadeia de caracteres, quase sempre devido à função *printf()*, que era reportada como vulnerável.

Metade das ferramentas está acima dos 50% de precisão, o que corresponde a um bom desempenho. Estas ferramentas são distinguidas pelas técnicas de análise mais sofisticadas.

[1] Para mais informação em, `http://spinroot.com/uno/`.

Ferramentas	Vulnerabilidades Detectadas	Vulnerabilidades Não Detectadas	Eficácia	Vulnerabilidades Incorrectamente Detectadas	Precisão
Crystal	13	251	4,9%	1	92,9%
Deputy	39	225	14,8%	31	55,7%
Flawfinder	49	215	18,6%	59	45,4%
ITS4	50	214	18,9%	167	23,0%
MOPS	10	254	3,8%	3	76,9%
PScan	5	259	1,9%	7	41,7%
RATS	50	214	18,9%	104	32,5%
Sparse	6	258	2,3%	2	75,0%
UNO	33	231	12,5%	14	70,2%

Tabela 1: Eficácia e precisão das ferramentas.

As ferramentas com valores abaixo dos 50% normalmente usam a análise de padrão, que é susceptível a reportar um número maior de falsos alertas. Em contrapartida, as ferramentas de análise por padrão conseguem localizar vulnerabilidades de um maior número de classes que as outras ferramentas, exibindo muitas vezes uma maior eficácia. As ferramentas que têm uma eficácia menor simplesmente detectam apenas uma categoria de vulnerabilidades.

4.4 Mute

Através da análise da secção anterior, é possível observar que juntando os resultados de todas as ferramentas, foi possível encontrar mais de metade das vulnerabilidades, o que é muito interessante porque o teste contém algumas "ratoeiras" para os analisadores. Como exemplo pode-se observar a Listagem 1; há uma vulnerabilidade do tipo *buffer overflow* na linha 5, devido à formatação na linha 2, onde se pode escrever para além do limite da estrutura de dados definida na linha 1; na linha 4 isso já não acontece. Este código obriga ao analisador a ter em conta a restrição de leitura de dados definida nas linhas 2 e 3, onde uma pode ler até vinte e cinco caracteres e outra até nove, quando faz a análise das linhas 4 e 5.

```
1 char str_scanf[10];
2 char fstr_untrusted[10] = "%25s";
3 char fstr___trusted[10] = "%9s";
4 scanf(fstr___trusted, str_scanf);
5 scanf(fstr_untrusted, str_scanf); // vulnerability
```

Listagem 1: Código com "ratoeiras".

Visto que as ferramentas separadamente não tiveram um desempenho muito satisfatório, a estratégia de combinar a detecção de erros tem o potencial de reportar mais vulnerabilidades que uma ferramenta só. Por esta razão, optou-se por incluir na construção do Mute todas as ferramentas que foram consideradas (ver módulo 1 da Figura 1). No entanto, se a agregação de resultados for efectuada de uma maneira simplista, pode-se cair na situação de se ter um número muito elevado de falsos positivos (logo, baixa precisão), o que reduziria em muito a utilidade da ferramenta.

Através dos resultados práticos, constatou-se que há algumas vulnerabilidades que nunca chegam a ser encontradas. Em concreto, o teste contém um total de 264 vulnerabilidades, tendo sido apenas detectadas 133. Estas 133 correspondem ao valor que o Mute no máximo

poderá detectar, ficando por encontrar 131, que correspondem aos falsos negativos. É de notar, que o Mute poderá localizar menos de 133 erros porque terá de descartar alguns avisos para reduzir os falsos positivos. No teste foram reportados 278 falsos positivos distintos.

Existiram também certas classes de vulnerabilidades que nenhuma ferramenta localizou, por não possuírem capacidades suficientes. As classes em questão foram: exceder o espaço de representação do inteiro (do inglês, *integer overflow* ou *underflow*), divisão por zero e ciclos infinitos. Se se retirassem estas classes ao teste (consistem em 60 vulnerabilidades), sobravam um total de 204 vulnerabilidades que as ferramentas podem detectar, tendo sido localizadas 133.

Uma vez que as ferramentas produzem um número considerável de falsos alertas, então deve-se usar técnicas para reduzir este número. A abordagem para combinar os relatórios das ferramentas é baseada em observar:

a) qual das ferramentas fez a detecção e;
b) quantas ferramentas localizaram a mesma vulnerabilidade.

Assim, foram considerados vários algoritmos ou métodos para tomar a decisão de se incluir ou não um aviso no relatório final. O **primeiro algoritmo** consistiu em atribuir um nível de confiança a cada ferramenta. Quando se tem a mesma vulnerabilidade detectada por ferramentas diferentes (possivelmente com confianças diferentes), é necessário decidir qual o valor é que deve ser associado à existência dessa vulnerabilidade. Foram considerados dois métodos diferentes, um baseado numa média em que se somam as confianças e divide-se pelo número total de confianças – *método 1*; e o outro baseado apenas na soma dos valores das confianças – *método 2*. Esta técnica também necessita da definição de um valor de limiar a partir do qual o Mute aceite o alerta. Este valor é usado para diminuir os falsos alertas, mas não deve impedir a apresentação das vulnerabilidades que realmente existem.

A partir dos testes realizados constata-se que há ferramentas que são melhores que outras para certas classes de vulnerabilidades. Como tal, considerou-se um **segundo algoritmo**, que em vez de atribuir uma confiança global às ferramentas, associa um nível de confiança que é especifico para cada classe de vulnerabilidade (ou seja, cada ferramenta passa a ter associado um vector de níveis de confiança). Os métodos de junção de confianças adoptados são semelhantes ao algoritmo anterior, mas desta vez designou-se por *método 3* à média, e *método 4* à soma.

As confianças atribuídas para o primeiro algoritmo têm por base os resultados da Tabela 1 na coluna precisão. Os valores utilizados no segundo algoritmo foram calculados usando a definição de precisão na equação 2, para cada categoria de vulnerabilidades. Em princípio, seria de esperar que o segundo algoritmo teria resultados mais favoráveis, uma vez que tira partido de informação mais detalhada de cada ferramenta.

O Mute foi concretizado de maneira a que fornecesse resultados com os dois algoritmos (e logo os quatro métodos). A cada configuração de parâmetros chamou-se um teste, que se encontra identificado por uma letra diferente na segunda coluna da Tabela 2. Na última coluna da tabela, é apresentado o valor de *limiar de confiança*, que decide se um aviso deve ou não ser descartado. O Mute só inclui no seu relatório os avisos para os quais tenham sido calculados valores de confiança superiores ou iguais ao limiar. Foi atribuído um limiar inferior de confiança de 0.3, umas vez que corresponde aproximadamente ao valor de menor precisão das diversas ferramentas. Utilizaram-se limiares superiores (exemplo, 0.5) para se perceber o comportamento das ferramentas no colectivo, ou seja, tipicamente mais que uma ferramenta teria de detectar a vulnerabilidade para que esta seja reportada. Não foram escolhidos os limiares de 1.0 e 0.0 porque não há qualquer ferramenta a detectar tudo com máxima precisão, além disso pretendemos reduzir o número de falsos positivos e não mantê-los.

O primeiro teste faz a uma simples agregação de resultados, sem se atribuir uma qualquer confiança às ferramentas, assim como ao limiar. Este teste corresponde ao algoritmo mais básico, e serve de base de comparação para as restantes configurações.

Teste		Método	Ferramentas									Limiar de Confiança
			Crystal	Deputy	Flawfinder	ITS4	MOPS	Pscan	RATS	Sparse	UNO	
1°	a)	-	-	-	-	-	-	-	-	-	-	-
2°	b)	1°	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,3
	c)	1°	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5
	d)	1°	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,7
	e)	2°	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,3
	f)	2°	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5
	g)	2°	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,7
3°	h)	1°	0,9	0,6	0,5	0,2	0,8	0,4	0,3	0,8	0,7	0,3
	i)	1°	0,9	0,6	0,5	0,2	0,8	0,4	0,3	0,8	0,7	0,5
	j)	1°	0,9	0,6	0,5	0,2	0,8	0,4	0,3	0,8	0,7	0,7
	k)	2°	0,9	0,6	0,5	0,2	0,8	0,4	0,3	0,8	0,7	0,3
	l)	2°	0,9	0,6	0,5	0,2	0,8	0,4	0,3	0,8	0,7	0,5
	m)	2°	0,9	0,6	0,5	0,2	0,8	0,4	0,3	0,8	0,7	0,7
4°	n)	1°	0,9	0,6	0,5	-	0,8	0,4	-	0,8	0,7	0,5
	o)	1°	0,9	0,6	0,5	-	0,8	0,4	-	0,8	0,7	0,7
	p)	2°	0,9	0,6	0,5	-	0,8	0,4	-	0,8	0,7	0,5
	q)	2°	0,9	0,6	0,5	-	0,8	0,4	-	0,8	0,7	0,7
5°	r)	3°	-	-	-	-	-	-	-	-	-	0,4
	s)	3°	-	-	-	-	-	-	-	-	-	0,6
	t)	3°	-	-	-	-	-	-	-	-	-	0,8
	u)	4°	-	-	-	-	-	-	-	-	-	0,4
	v)	4°	-	-	-	-	-	-	-	-	-	0,6
	w)	4°	-	-	-	-	-	-	-	-	-	0,8

Tabela 2: Testes realizados o Mute.

O segundo teste baseia-se numa simples atribuição de uma confiança em comum para todas as ferramentas, afirmando desta forma que todas têm um quanto de confiança e encontram-se no mesmo patamar. Quanto ao valor do limiar de confiança, esse teve que variar, para daí analisar os métodos empregues nos testes.

O terceiro teste atribui uma confiança específica para cada ferramenta, uma vez que cada uma tem capacidades diferentes, causando que de certa forma os falsos alertas das ferramentas menos precisas fossem eliminados.

O quarto teste foi realizado sem duas ferramentas pela razão que estas têm o maior número de falsos positivos (ver Tabela 1). Logo, será do interesse perceber o comportamento do Mute sem o ITS4 e RATS. Neste teste, foi retirado o limiar de 0.3 para garantir que a ferramenta com pior precisão não era capaz sozinha de impor a aceitação dos seus avisos.

O quinto teste associou a cada ferramenta e classe de vulnerabilidades um nível de confiança (estes valores não estão representados na tabela). Esta configuração acaba por fazer a atribuição mais fina das confianças.

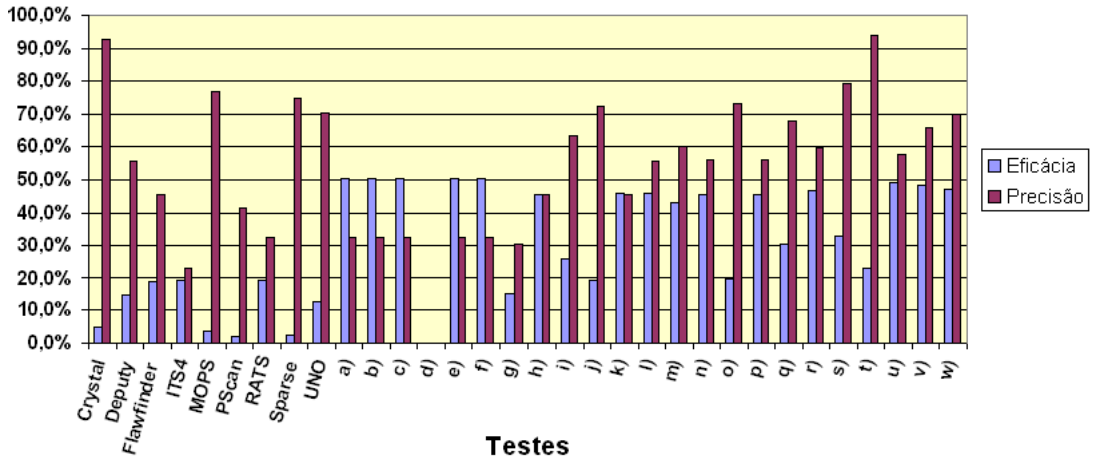

Figura 3: Eficácia e precisão do Mute nas várias configurações.

No Gráfico 3 apresentam-se os valores de eficácia e precisão das ferramentas individuais e das várias configurações do Mute. Para o Mute, observa-se um crescimento da precisão a partir do teste h), o que é de esperar uma vez que as ferramentas deixaram de ter uma confiança estática de 0.5 e passou-se a usar uma confiança mais exacta para cada ferramenta.

Nos primeiros testes [a), b), c), e), f)], observa-se uma eficácia a 50% que corresponde ao máximo de vulnerabilidades que o Mute poderia detectar. O teste a) corresponde simplesmente à junção de vulnerabilidades em comum com outras ferramentas, onde temos uma alta eficácia, mas baixa precisão. No teste b) e c) obtém-se um resultado semelhante porque todas as ferramentas têm a mesma confiança. No teste d), como o limiar era de 0.7 e o máximo de confiança a obter era de 0.5, todas as vulnerabilidades foram rejeitadas. Os resultados do teste e) e f) também eram esperados por causa do valor de confiança depositado em cada ferramenta de 0.5. Os resultados do teste g) são explicados porque existem certas vulnerabilidades que não são detectadas por todas as ferramentas, daí ser impossível em alguns casos obter um valor superior ao limiar de 0.7. A partir do teste h) esperava-se uma melhoria dos resultados, mas não se sabia em quanto. Os melhores resultados acabaram por ocorrer nos últimos testes, onde se detectou praticamente todas as vulnerabilidades e se teve uma precisão elevada.

Comparando com os resultados das ferramentas individualmente, o Mute é claramente melhor, quer em precisão como em eficácia. Não é possível comparar, cada métrica separadamente, uma vez que se torna a comparação desigual. A única ferramenta que ultrapassou claramente o Mute em precisão foi o Crystal, no entanto, isto é conseguido com baixa eficácia.

As duas técnicas que foram utilizadas para diminuir o número de falsos positivos são diferentes em vários pontos, sendo uma mais eficaz que outra. Ou seja, demonstrou-se ser mais interessante atribuir uma confiança para cada classe de vulnerabilidade em cada ferramenta, em vez de se ter uma confiança global por ferramenta. Esta solução tem como consequência que ferramentas com baixa confiança não conseguem por si só forçar a publicação de certas vulnerabilidades, necessitando que outras ferramentas também as encontrem. Além disso, há ferramentas que para um tipo de vulnerabilidades têm maior precisão que outras.

Em resumo, o Mute no máximo pode detectar 133 vulnerabilidades, uma vez que se encontra dependente do trabalho realizado pelas outras ferramentas. Pode-se observar que com base nas duas métricas (eficácia e precisão) os testes u), v) e w) deram os melhores resultados, conseguindo obter no máximo uma eficácia de 50% (que equivale aproximadamente às 133 vulnerabilidades) e uma alta precisão, onde se tem menos de 75%[1] de falsos positivos. Com estes valores é de considerar que o Mute é bastante eficaz no seu funcionamento.

5 Conclusões

Ano após ano, o número de vulnerabilidades que têm sido descobertas nas mais variadas aplicações tem vindo a aumentar. É possível apontar diversas causas, entre elas a complexidade dos programas, a falta de formação ou cuidado dos programadores, e interfaces mal especificadas.

Os analisadores estáticos são eficazes porque conseguem detectar um conjunto de classes de vulnerabilidades, com relativa rapidez e eficácia. As ferramentas de análise estática são úteis para os programadores porque possibilitam a correcção de um número substancial de vulnerabilidades antes do código entrar em testes ou ser disponibilizado para os utilizadores.

[1] Através de uma combinação simples dos resultados das ferramentas obtem-se 278 falsos positivos, ao passo que com os últimos testes u), v) e w) obtem-se os seguintes valores de falsos positivos: 96, 66 e 53. A partir daqui é possível observar que se reduziu no pior caso 75%, mas nos restantes ainda mais.

Foi proposta uma maneira de se avaliarem e compararem estas ferramentas, por intermédio de um teste que contém código vulnerável. O teste considera várias classes de vulnerabilidades, ou seja, foram adicionadas vulnerabilidades e código não vulnerável para o mesmo tipo, e com algumas vulnerabilidades camufladas. Isto serve para perceber se o analisador estático consegue detectar as vulnerabilidades e compreender as suas dificuldades.

Depois de estudar e avaliar as capacidades das ferramentas, procedeu-se à construção do Mute. No Mute foram consideradas duas técnicas de análise/filtragem de vulnerabilidades, uma onde é depositada a confiança de cada ferramenta, e outra que consiste em atribuir uma confiança para cada classe de vulnerabilidade. A avaliação do Mute demonstrou a utilidade de se usarem métodos de agregação, tendo ele conseguido localizar mais vulnerabilidades que as ferramentas individualmente, mantendo os falsos positivos com valor reduzido.

Referências

[1] Dominique Alessandri, *Attack class based analysis of intrusion detection systems*, Ph.D. thesis, University of Newcastle, School of Computing Science, May 2004.

[2] Brad ArKin, Scott Stender, and Gary McGraw, *Software penetration testing*, IEEE Security & Privacy **3** (2005), 84–87.

[3] Hao Chen, Drew Dean, and David Wagner, *Model checking one million lines of c code*, Proceedings of the 11th Annual Network and Distributed System Security Symposium, Feb 2004, pp. 171–185.

[4] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gray, and George Necula, *Dependent types for low-level programming*, Proceeding of the 16th European Symposium on Programming, March 2007.

[5] Gerard J. Holzmann, *Static source code checking for user-defined properties*, Proceeding of the 6th World Conference on Integrated Design and Process Technology, 2002, pp. 26–30.

[6] Marksim Orlovich and Radu Rugina, *Memory leak analysis by contradiction*, Proceedings of the 13th International Static Analysis Symposium, Springer, August 2006, pp. 405–424.

[7] Alexander Ivanov Sotirov, *Automatic vulnerability detection using static source code analysis*, Ph.D. thesis, University of Alabama, 2005.

[8] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw, *Its4: A static vulnerability scanner for c and c++ code*, Proceedings 16th Annual Computer Security Applications Conference, IEEE Computer Society, December 2000, pp. 257–269.

[9] John Viega and Gary McGraw, *Building secure software: How to avoid security problems the righ way*, Addison-Wesley, 2006.

[10] Marco Paulo Amorim Vieira, *Testes padronizados de confiabilidade para sistemas transaccionais*, Ph.D. thesis, Universidade de Coimbra, Julho 2005.

[11] Yves Younan, Wouter Joosen, and Frank Piessens, *Code injection in c and c++: A survey of vulnerabilities and countermeasures*, Tech. Report CW 386, Katholieke Universiteit Leuven from Department of Computer Science, July 2004.